Create an Amos arcade game.

DISK ENCLOSED

AC's TECH / AMIGA

For The Commodore

Volume 3 Number 2
US $14.95 Canada $19.95

Olé

- ARexx Disk Cataloger
- Getfile Shell for True BASIC
- Olé–Amos arcade game
- Programming the Amiga in Assembly Language Part VI
- Porting a B+Tree Library
- Assembly Language Computer Simulations
- Wrapped Up in True BASIC

02
0 74470 78640 5

ON WITH THE SHOW!

The 4th Annual
WORLD OF COMMODORE AMIGA
IN NEW YORK CITY

April 2, 3 & 4, 1993

Come to America's greatest exhibition and sale of Amiga hardware, software and accessories!

SEE, TRY AND BUY — ALL THE LATEST EXCITING PRODUCTS:
The dazzling new Amiga 4000!
The Amiga 1200!

FREE SEMINARS with show admission
Desktop Video! Desktop Publishing!
Multimedia! Animation & Graphics!
AND MUCH, MUCH MORE!

New York Passenger Ship Terminal, Pier 88
(Between 48th & 52nd on Hudson River)

Friday & Saturday 10 a.m.–5 p.m.
Sunday Noon–5 p.m.

ADMISSION: $15.00 per day, $30.00 for three-day pass.

SHOW HOTEL: Holiday Inn Crowne Plaza, 1605 Broadway, New York, NY 10019. For reservations call (212) 977-4000. Show rate $135 single or double. Deadline March 9, 1993.

For more show information, phone (416) 285-5950.

SAVE WITH PRE-REGISTRATION

To pre-register complete and mail this form with check for $25.00 (3-day pass) or $10.00 (single day) BEFORE MARCH 5, 1993.

Name_____

Address_____

City_____ Zip_____

Make check payable to RAMIGE MANAGEMENT GROUP.
Mail to: 3380 Sheridan Dr., Suite 120, Amherst, NY 14226

Contents

Volume 3, Number 2 AC's TECH/AMIGA

Departments

```
printf("Hello");

print "Hello"

JSR printMsg

say "Hello"

writeln("Hello")
```

Whatever language you speak, AC's TECH provides a platform for both gaining insight and sharing information on its most innovative implementation for the Amiga. Why not see if your latest programming endeavor can help a fellow Amiga user expand upon his or her vocabulary? To be considered for publication in AC's TECH, submit your technically oriented article (both hard copy & disk) to:

AC's TECH Submissions
PiM Publications, Inc.
P.O. Box 2140
Fall River, MA 02722-2140

AC's TECH/AMIGA

ADMINISTRATION

Publisher:	Joyce Hicks
Assistant Publisher:	Robert J. Hicks
Administrative Asst.:	Donna Viveiros
Circulation Manager:	Doris Gamble
Asst. Circulation:	Traci Desmarais
Traffic Manager:	Robert Gamble
Marketing Manager:	Ernest P. Viveiros Sr.

EDITORIAL

Managing Editor:	Don Hicks
Editor:	Jeffrey Gamble
Hardware Editor:	Ernest P. Viveiros Sr.
Senior Copy Editor:	Paul Lamvee
Copy Editor:	Elizabeth Harris
Video Consultant:	Frank McMahon
Illustrator:	Brian Fox

ADVERTISING SALES

Advertising Manager: Wayne Amada

1-508-678-4200
1-800-345-3360
FAX 1-508-675-6002

AC's TECH For The Commodore Amiga™ (ISSN 1063-7079) is published quarterly by PiM Publications, Inc., One Current Road, P.O. Box 2140, Fall River, MA 02722-2140.

Subscriptions in the U.S.: 4 issues for $44.95; in Canada & Mexico surface: $52.95; foreign surface for $56.95.

Application to mail at Second-Class postage rates pending at Fall River, MA 02722.

POSTMASTER: Send address changes to PiM Publications, Inc., P.O. Box 2140, Fall River, MA 02722-2140. Printed in the U.S.A.

First Class or Air Mail rates available upon request. PiM Publications, Inc. maintains the right to refuse any advertising.

PiM Publications inc. is not obligated to return unsolicited materials. All requested returns must be received with a Self Addressed Stamped Mailer.

Send article submissions in both manuscript and disk format with your name, address, telephone, and Social Security Number on each to the Editor. Requests for Author's Guides should be directed to the address listed above.

AMIGA® is a registered trademark of Commodore-Amiga, Inc.

Printed in the U.S.A.

Startup-Sequence

I'd like to first address a letter that I have received since the last issue. The first is from Raymond Zarling and is in regards to the article "Trading Commodities in Workbench 2.0," *AC's TECH Volume 3, Number 2*. He writes:

> The article "Trading Commodities in Workbench 2.0" was very interesting. It offered a focused way for me to learn more about writing commodities, which is one of those things I had been meaning to get around to for some time. I liked the writing style, too, which was helpful in explaining what was happening without becoming condescending.
>
> But the code, and hence the exposition, contained a bug. It took me some time to track it down, and it had meanwhile become embedded in another commodity I wrote myself modelled on what I had learned from this article. Lest other of your readers similarly repeat this mistake, I thought I would write in the hope you could publish a correction.
>
> The bug manifests itself if the user specifies a CX_POPKEY tooltype for the commodity. The code searches the tooltypes for this, and assigns the resulting string to a character pointer x, near the middle of function cxparse(). Then ArgArrayDone() is called, and that is a mistake, because the string to which x points is freed by that action, so the popkey string could be overwritten by the time it is used in the AddKey() call. As a result, about half of the time when MMB_CX started in my startup-sequence (while there were plenty of other commodities starting up at the same time) the installation would fail with a "broker error".
>
> According to Commodore's autodocs for ArgArrayDone(), in cx_lib.doc:
>
> ```
> void ArgArrayDone (VOID)
> ```
>
> This function frees memory and does cleanup required by ArgArrayInit(). Don't call this until you are done using the tooltypes argument strings.
>
> One way of fixing the problem would be to copy the string x to some memory owned by MMB_CX before calling ArgArrayDone(). When I made that change and recompiled, the random errors I had been getting when using MMB_CX in my WBstartup ceased.
>
> Thank you for publishing an otherwise excellent article.
>
> Sincerely,
> Raymond L. Zarling

I would like to thank Mr. Zarling for bringing the bug to our attention and also thank him for offering a solution.

Disabled Users

While at the WOC A Toronto, I was approached by an Amiga user wanting to know if I knew of any good public-domain programs for the handicapped. This gentleman was confined to a wheelchair. Specifically, he was looking for a program that would allow him to use his Amiga to control electrical appliances in his house and anything else that might make his life easier. I directed him to *AC's GUIDE* and its complete list of Fred Fish software. On looking into the guide sometime later, I realized that there just is not a great deal of software available for handicapped Amiga users. The same holds true for the other major platforms. This is primarily because computer software programs designed for the physically and developmentally impaired are usually highly specialized and directed to the needs of an individual and not a group of users.

But does the Amiga have the potential to reach out to this special group of users? Given the power and flexibility of the platform, combined with the wide range of development packages available, the answer should be yes.

Many handicapped persons work with computers regularly. I have a friend, Julie, who is autistic. Julie has used computers at school and at home. She has a Nintendo Entertainment System® that she has mastered and can play Super Mario Brothers!® better than Tommy played pinball. Julie has used my A600 on occasion. She is comfortable with using a mouse and is able to navigate around Workbench with no more trouble than the average user. But once she's on the computer, what will she do with it? Well, she enjoyed *DeluxePaint IV* and had a ball with *Pro Write*'s speak function. But aside from playing games, that was it.

Julie, as well as other disabled persons, can greatly benefit from the use of computers. We need more commercial and public domain software geared to the handicapped. There are specific areas where Amiga software could be beneficial, but development should not be limited to those areas. Business, productivity, entertainment, all categories should be included, not just educational and developmental.

The Challenge

AC's TECH is sponsoring a development contest. Basically, you have to develop an Amiga application geared toward disabled users. You may use any development system you wish and you may address a specific disability or present a program for a general handicapped audience. This will require some work. The time you spend on the application, both in research and development, will be time well spent. You will be doing a world of good for the community of disabled users.

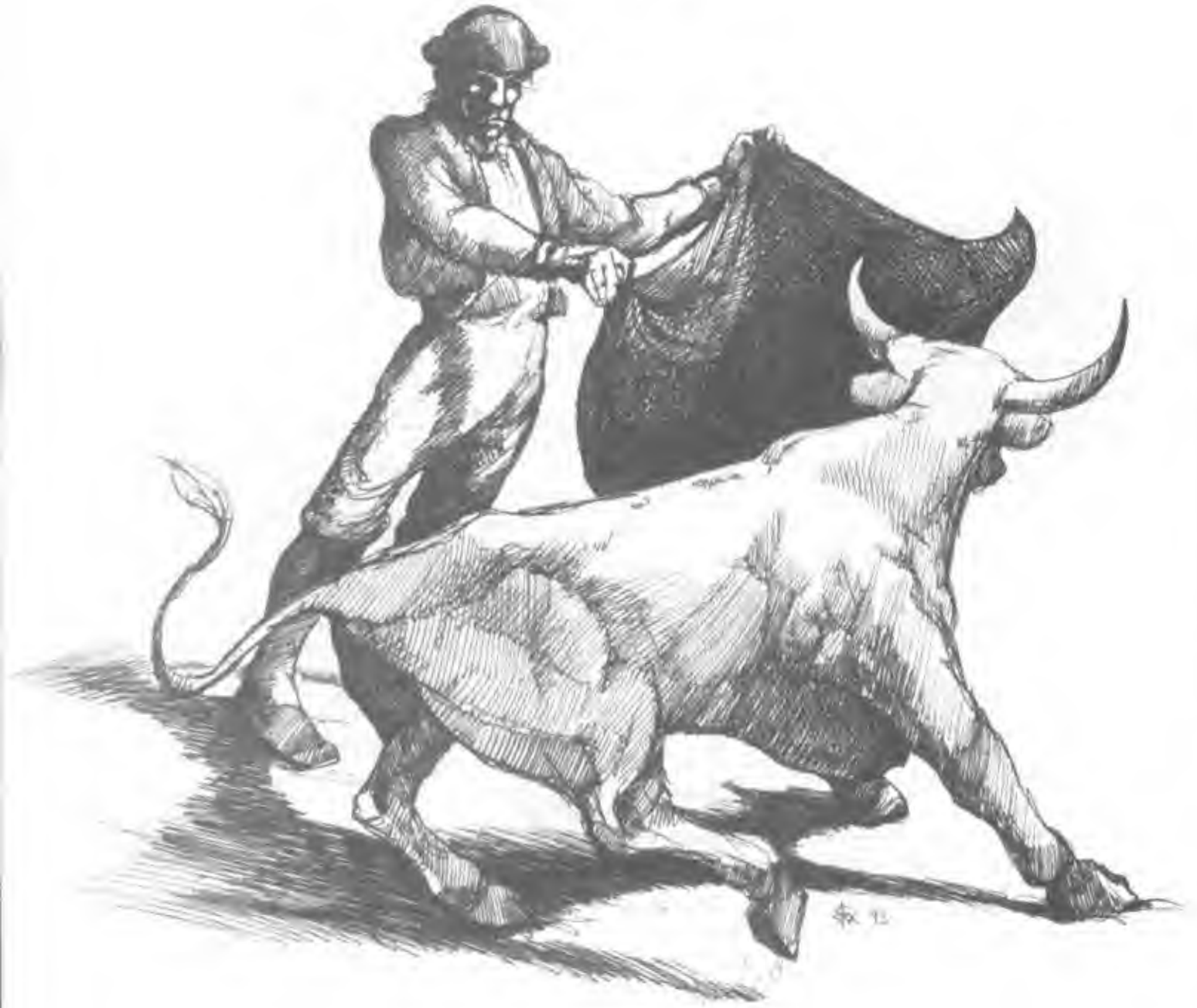
AC's TECH will award prizes for the best application and two runners-up. The entry deadline is October 22, 1993. Your entry should be either a fully functional program or a working demo of the program. You must provide documentation with the program. Contest winners will be announced in the 4.1 issue of *AC's TECH*. We have put together a complete set of contest rules and entry information. We have also assembled a list of guidelines to follow and suggestions for possible development. It costs nothing to enter the contest and we are hoping for a large turnout. If you like to write code and develop applications, this is a good project to turn your talents and time toward.

Jeff Gamble
Editor

For the Contest Information Pack call or write *AC's TECH* at:

AC's TECH Contest
P.O. Box 2140
Fall River, MA 02722-2140
1-800-345-3360

OLÉ!



An Arcade Game Programmed in AMOS BASIC

by Thomas J. Eshelman

This article is intended to help the reader more fully appreciate an amazing language dedicated exclusively to the Amiga computer. A close examination of the accompanying code will reveal AMOS's simplicity as well as its power. We will carefully dissect a generic Amiga arcade game written in AMOS. AMOS is not dedicated to games programming, rumors to the contrary notwithstanding. However, because of its extremely high horsepower in the realms of sights and sounds, it is a natural preference for anyone thinking about programming games. Similarly, it can easily toss off those exercises commonly referred to as "Eurodemos." We will pay special attention to a few of the more important and powerful components of the language sometimes ignored in other discussions. After realizing just how little code has been written to realize a fairly complete gameimmersed as it is in copious comments, I hope that you will conclude it is worth your while to invest in AMOS. A compiled version of *OLÉ* is available for those of you without an AMOS interpreter.

What is AMOS?

AMOS is a superset of the BASIC programming language. Several hundred instructions, tailored especially to engage the special hardware of the Amiga computer, are appended to a garden variety BASIC. This is done in a fashion friendly to multi-tasking and without fear of invoking the dreaded Guru. The terms "AMOS" and "BASIC" may be and are used interchangeably.

Along with this BASIC interpreter comes a host of accessory programs, including a Sprite and Bob Designer/Editor, an AMAL Language Editor, a background editor called "TAME," an unusually powerful Menu Editor (like animated MenuItems!) and other programs that convert IFF, SMUS, Tracker, etc., files into formats directly useable by the interpreter. All these powerful utilities are themselves written in BASIC! A BASIC compiler and a 3-D object generator-animator are also available but at extra cost. One wonders what might have transpired had the Amiga been bundled with programs such as Power Windows™, AMOS and the ARP library at its debut in 1985.

OLÉ—The Game

OLÉ is a version of John Gilmore's clever PD game titled, *Fast Amigo*. The player moves the matador across each of four decks, picking up prizes as he climbs ladders leading to the next higher decks. All the while he must jump over charging bulls to avoid being gored. Appropriate sound effects are rendered. After the four decks are conquered, the user advances to the next skill level. With each of the nine skill levels, the matador becomes more tired, and hence slower. The bulls, however, run faster as they get madder, making them more difficult to avoid.

The original game was a bit fast for many mere mortals, even when run on a 68000 Amiga, unless you happened to be part mongoose. On an accelerated machine, it nearly defied visibility. We want to illustrate AMOS, however, not reinvent the wheel. Fast Amigo exhibited a great sense of humor while it kept the plot simple. OLÉ will slow down the action not because of its BASIC origins, but rather by making the vertical blanking interval serve as an internal timer. Let's start by examining a less familiar but important feature of AMOS.

Memory Banks

The term "memory bank" has quite an exotic "ring" to it, but the concept is actually very simple. Memory banks are merely areas of memory which are automatically incorporated into the main file at such time as the file is saved to disk. Most of the time, banks are created automatically by BASIC, totally tranparently to the user. However, means are provided for the more advanced programmer to create and access them as required for some special program application. If I paint some Bobs in the Sprite Editor and save them to disk, they will be saved as an ".abk" file by default. When I load this ".abk" file using an instruction written into my BASIC program, memory bank number "1" will be created, and the image data will load there.

The purpose of memory banks is to hold data, such as images, music, and sound samples. AMAL programs are the exception (more on them later). It is logical enough to provide separate storage areas for the many categories of audio/visual data when you expect to commonly deal with them in large quantities. Thus, the BASIC interpreter is spared from having to gather it together from bits and

pieces scattered throughout memory. To get a quick feel for the subject, let's look at some memory banks in action.

Table 1 reveals 8 lines of code the programmer may choose to run but once, as long as he doesn't mind working with a large file. If he chooses to save the file after he runs it, these lines may thereafter be deleted! The first two lines each allocate a screen and load an IFF file, created for example from DeluxePaint™, from the disk into it. The next four lines are more directly on point. They automatically allocate and load memory banks with .abk (amosbank) files, the particular ID numbers of the banks being supplied as defaults by BASIC. You can't get much simpler than this!

Note that we create ".abk" files whenever we generate output from any of the various accessory or format-conversion programs, such as the Sprite Editor or the Sample_Bank_Maker.

Finally, we take those IFF screens and compress them into memory banks, this time employing the user's choice of bank ID numbers, with the "Spack" (screen pack) command. Spack allocates a memory bank of sufficient size to contain the compacted IFF file, compacts it and copies it thereon. The purpose behind compacting the files is to save disk space, of course. The fruits of these operations are not realized, therefore, until we "save" the program. On that occasion, the screens will be saved to disk in a compacted form.

You should also be aware that there are a total of 15 of these memory banks available. The first five should be used by the programmer only in a pinch, since BASIC will also use them by default for specific data types. It is legal to append to other data within a bank, but why ask for confusion as to "what is where" until after all 15 have been used? Here banks 6 and 7 were chosen because banks 1, 3, 4, and 5 are "spoken for." Screens 0 and 2 are compressed, and shoehorned into the banks with only these two lines of code!

To make all this data an integral part of the program, we simply run BASIC and then resave the file! A 9000 byte file may "blossom" suddenly to one of 200,000 bytes, especially where a number of sound samples exist in a bank. Sound sample files are large by their very nature. Of course, we can subsequently delete the lines of code in Table 1 since we need never load these files again, unless we choose to modify them. Nor must these files be carried separately elsewhere on the disk along with your BASIC file. All the code, and all the data are now one "chunk," as it were.

We can then replace the Table 1 code with that in Table 2, if desired. When the disk file is loaded, AMOS automatically remembers, creates, and loads the same banks that were previously used to save the data! As far as the IFF pictures are concerned, the Unpack commands cause the banks containing them to be decompressed, screens to be allocated for them, and the pictures then copied from the banks into the screens. The banks may thereafter be erased if desired. This would free their memory for other uses.

The bottom line is that the choice is yours whether to carry around a number of small files, or to weld them into one large file. The latter includes the infamous memory banks as integral parts. For development work, I find it convenient to work with the large file, but I never delete the "Load" commands. Thus, any changes I make to an ".abk" file will be reflected in the dominant file without my having to remember to resave it separately. If you program on an unaccelerated machine, or worse, with no hard drive, it will certainly pay to work with the small, individual files. No time will then be wasted reloading or saving unaltered portions of your program.

What is AMAL?

The Amiga Animation Language. That's what! AMAL is a small sublanguage. In the AMOS system, the BASIC interpreter becomes and functions as a compiler of AMAL instructions! The sOLE purpose of AMAL is to move and animate any graphic elements, including Sprites, Bobs, screens, and rainbows.

The use of AMAL is optional, however. It contains only two instructions, irrelevant to OLÉ, that are not readily duplicated in BASIC. On the other hand, AMAL instructions have the advantage of being as fast as anything written in C or assembler, being as they are part of a compiled language. AMAL programs run asynchronously with their BASIC parent. They run at their own very high speed.

AMAL may be written using either the AMAL Editor, the output of which is automatically incorporated into memory bank no. 4 as an ".abk" file, or it may be written directly in the BASIC editor along and in line with other, common BASIC instructions.

AMAL consists of only 10 commands but these are very finicky as to their formatting. They consist of one (rarely two) uppercase letters only. Lowercase is not recognized. For example, to assign a value to a variable, there exists a command, the use of which is mandatory, called, "Let". This may just as well be written, "L.", but it is error to write it as "LET", "LeT" or "lET".

Table 1

Load Iff "Title.pic",2	Allocate screens.
Load Iff "OLE.pic",0	Load IFF files into them.
Load OLESprites.abk	Create and load bank 1.
Load OLÉAmal.abk	" " " bank 4.
Load OLÉSamples.abk	" " " bank 5.
Load OLÉMusic.abk	" " " bank 3.
Spack 0 To 6	Create banks 6 and 7. Compress
Spack 2 To 7	screens 0 and 2 into them.

Table 2

Unpack 7 To 2	Allocate screen 2. Copy bank 7
Erase 7	into it. Then erase the bank.
Unpack 6 To 0	As above. Decompress bank 6
Erase 6	into screen 0, and erase bank.

Note: One never need refer to any ".abk" file as such.

By default, there are 16 AMAL programs or channels, each of which is driven or started up by its own hardware interrupt. Each channel animates and moves one graphics object. They all run independently of BASIC, and neither interferes with nor slows down the other. Since BASIC is not in the same league as any compiled language when it comes to speed, Francois Lionet, the father of AMOS, intends we do our animations in AMAL and our screen setups and other grunt work in BASIC. We may call an Amal program but once from BASIC, and not consider it again. The Amiga's interrupt system takes over.

There are two situations, however, where we will want to call AMAL programs out of the BASIC code itself, rather than letting them run by themselves via interrupts. First, we may want to simultaneously animate more than 16 Bobs or computed Sprites. The Amiga's hardware restricts us to using only 16 interrupts. BASIC can set up and run any number of channels, however. The default setting, as a matter of fact, is 64 Bobs or Sprites!!

Secondly, we can never do collision detection within an interrupt handler. The Amiga's Blitter hardware prohibits this. We still want to do collision detection, however, from within AMAL and not BASIC, since AMAL is always faster by several orders of magnitude, even when called from BASIC.

The AMOS system provides a workaround in the form of the "Synchro" command. With each call to Synchro, all the AMAL programs are launched asynchronously with the rest of the BASIC code. This gives us the best of both worlds. You arrange to call the Synchro command once before each vertical blanking interval (VBI). This is the route we will go in OLÉ, and this is the reason we broach the subject of AMAL this early.

Writing AMAL programs

Let us examine some AMAL code. When writing AMAL instructions, I humbly suggest you utilize the AMAL editor as opposed to writing them directly in BASIC. It is legal to do both, but the first scheme avoids the confusion offered by the interminably long columns of inline code characteristic of BASIC, and it saves having to repeatedly type the damnable 'A$=A$+"..." line starter. Also, the AMAL Editor allows instant testing of just your AMAL channel, as long as you bother to load a few Bobs or Sprites into the "Environment Channel" for display purposes. AMAL also provides a debugger with which you can read, write, or follow AMAL variables. The down side of all these wondrous artifacts is that you must learn to use the AMAL Editor. Please turn your attention to Table 3.

Here is an AMAL program consisting of seven commands in four lines. However, it is enough to continuously gallop one of our bulls back and forth across the display. The same code appears in channels 2, 3, and 4, each controlling one bull. "Let" is our first AMAL command. Observe it appears in both legal formats mentioned above. 'X' is a "magic" variable that always contains the horizontal coordinate of whatever you are animating with this particular channel. It is a local variable with that name, "X", hardcoded. Each channel has its own "X". The value "15" is chosen since it starts the bull just inside the left edge of the screen. "Y", obviously, is the vertical coordinate and works just as described for "X". "A:" is a label. Note well the distinguishing colon character!

"Anim" is another of our 10 AMAL instructions. It will display image #4 (a standing bull) for nine ticks (50ths of a second), then image #5 (a stretched out bull) for 15 ticks. The value "0" directs it to cycle continuously. "Move" is yet another of the 10 AMAL commands. Here, we move right 290 pixels, down 0 pixels in "RA" time quantums. "RA" is one of 26 global AMAL variables (RA-RZ) that we may access from within BASIC as well as from within any other AMAL program! They work much like C globals in the latter language. By reducing the value in RA, we quicken the horizontal speed of the galloping bull. Increasing RA does the opposite.

The next Anim and Move commands are very similar. We exhibit mirror images (nothing to it thanks to the Sprite Editor) so our bull faces left, and run him left for 290 pixels. 'Jump A' does just as you would expect. It loops forever. 'Jump' could have been written merely as 'J'. The bull ordinarily won't stop galloping back and forth across the screen until we quit the program or turn off this AMAL channel.

Table 3

```
' chan 1 to bob 1
' bottomost bull

Let X=15:          L Y=196:

A: Anim 0,(4,9)(5,15); Move 290,0,RA:

A 0,(2,9)(3,15); M -290,0,RA:

Jump A:
```

Table 4

```
'chan 8 to bob 8
'top prize

'assign a value from setprizes procedure in basic
' to this channel's x

      Let X=RK:

'if collision legal, check for same.
'else pause and retest flag

      A: If RL=0 Jump B: Pause: Jump A:

'if a collision is detected, set anti-repeat flag

      B: If BC(8,0,0) J C: P: J A:

      C: L RL=1:

'display appropriate prize image.
'increment score per current skill level

   L A=RC+19: L R0=RC+1: L RM=R0*10+RM: L RP=1: J A:
```

Table 5

```
' chan 0 to bob 0
' the matador

' r0=loop counter, r2=climbing flag, r8=jumping flag
' ra=bull speed, rd=operations deck, rm=matador speed
' re, rf and rg=ladders centers
' d=prize collision loop preventer

L R0=0; L R2=0; L R8=0;

' read the joystick - synchronize to display
' order = fire, left, right, up

A: Pause;
B: If J1&16 Jump F;
C: If J1&4 Jump G;
D: If J1&8 Jump H;
E: If J1&1 Jump I;

' lastly, check if a moving bull collided with
' a stationary matador

If BC(0,1,4) J P; J A;

' if not already jumping, set the 'i am jumping' flag,
' jump quickly, pause for a period whose length is
' relative to the bull speed, check for collisions with
' bulls while slowly descending

F: If R8=1 J A; L R8=1; L Y=Y-25; F R0=1 T RQ N R0;
   F R0=1 T 25 If BC(0,1,4) J P; L Y=Y+1; N R0;
   L R8=0; J A;

' return to reading joystick if move would cause clipping

G: If X > 8 J J; J A;
H: If X < 312 J K; J A;

' test the 'i am climbing' flag, reset jumping flag,
'and select proper ladder

I: If R2=1 J A; L R8=0;
   If RD=0 J L; If RD=1 J M; If RD=2 J N;

' pick left or right image, reset flags,
' move matador using f to n loop - helps render cpu
' speed irrelevant - also check for collisions

J: Let A=0; L R2=0; L R8=0; F R0=1 T 2 L X=X-RN;
   If BC(0,1,4) J P; N R0; J A;

K: Let A=8; L R2=0; L R8=0; F R0=1 T 2 L X=X+RN;
   If BC(0,1,4) J P; N R0; J A;

' determine the proximity of the matador's hot spot with
' pertinate ladder's - then execute one of the following
' L L R7=RE-X; J O; M: L R7=RF-X; J O; N: L R7=RG-X; J O;

' see if hot spots are within 6 pixels - pick the frontal
' image - climb 50 lines slowly - this will remain
' constant regardless of cpu because of the vblanking
' implied in the for loop - every second pixel of climb,
' check for collision with bull - set climb flag,
' increment operations deck level number, reset the
' 'got prize' flag
'
' heads up..heads up.. note how we assign bool expressions
' individually to r6 and r9, check both simultaneously in
' one 'if' test - this saves one jump instruction, keeping
' within the 5 loop limit.

O: Let R6=R7>6; Let R9=R7 < -6; If R6|R9 J A; Let A=7;

   For R0=1 To 25 Let Y=Y-2; If BC(0,1,4) J P; N R0;

   Let R2=1; Let RD=RD+1 Let RL=0; J A;

' matador collided with a bull - move him off screen,
' set 'gored' flag

P: F R0=1 T 15 L X=X-15; L Y=Y-15; N R0;

   L R2=0; L R6=0; L R8=0; L R9=0; L RO=1; J A;

end AMAL
```

from within BASIC. In our situation, we could also stop the action simply by failing to recall "Synchro".

This is a good time to note a point which the otherwise extensive manual does not. Both the apostrophe character and the asterisk can be used to create a comment line when in the AMAL Editor. What is not mentioned, however, is that *no uppercase characters* may appear in the comment itself!

Let's study another AMAL channel. This one is dedicated to prize collection. There are four of these also: 5, 6, 7 and 8. The prizes only 'move' when skill levels are reset. However, we want to handle them within AMAL because AMAL presents a very convenient method for detecting collisions. When a collision is detected, we want instant substitution of a score image for the prize image. Table 4 is part of what is seen when the file, OLÉAmal.abk, is loaded into the AMAL Editor. In only five lines of code, a prize bob is fixed, collision with the matador bob is checked, its image swapped on the display if a collision were detected, and the score incremented in accordance with the current skill level!

"RK" is the horizontal coordinate assigned to Bob 8 in BASIC. You will see this is a randomly generated value. Unless we are currently colliding, we jump to label B. Else, we will wait for a vertical blanking

interval, and reloop back to label A. "If", is another AMAL command. Here we see that the *only action available if* an expression evaluates as *true* is to *jump*! This is an AMAL idiosyncracy that took me quite a while to become accustomed to.

"Pause," another AMAL command, performs the same function as WaitTOF() does in C. It synchronizes the execution of the program with the vertical blank interval, the same time quantum in any and all Amigas. At label B:, we call the AMAL Bob collision tester function, BC(), on this Bob, with the matador (channel 0). If none, pause and reloop. Else, set a flag to prevent a collision loop.

The 'A' on the last line is the third AMAL 'magic' variable. It refers to the image number for this Bob as it is found in the Sprite Bank, or bank #1. "RC" is a value we set in BASIC to represent the current "skill level." For example, if this line of code is reached, the "Bag of Gold" prize will suddenly become a large "10" on the display. If you were to load the file, "OLESprites.abk" into the Sprite Editor, you will observe each image as you punch in an image number. It follows that this is how you edit them.

"RM" is the score printed on title bar. The algorithm seen here increments the lowest level by 10 with each prize, and the highest level by 90. Note that in AMAL, all operations evaluate strictly left to right! Parenthesis are illegal! The "RP" variable is used only in channel 8 (the uppermost deck). When RP is set, it tells BASIC that the user has attained the top deck, so that BASIC will bump him to the next higher skill level.

Finally, let's examine the Amal channel dedicated to operating the matador figure in response to the joystick. Begin by giving your attention to the comments contained within the code in Table 5. This is now becoming self- evident to you, and much less external explanation is required.

Remarks Regarding The Matador Channel

In addition to the previously mentioned 26 global variables, "RA-RZ", that may be shared with BASIC and all other AMAL channels, this code illustrates the existence of 10 additional local AMAL variables that exist in and are private to each AMAL channel in use. These also bear hardcoded names "R0-R9", and like the RA-RZ globals, may contain any 16-bit signed integer.

Thus far, we have already learned seven of the AMAL commands: Let, Move, Anim, Jump, If, Pause, and For.

Among important things to note are the 'For' loops. "For...To...Next" is abbreviated as F, T, and N. AMAL automatically does a WaitVbl, or Pause, with each iteration of an F T N loop! This causes not only remarkable smoothness to the animation, but helps render the CPU speed irrelevant! The liberal use of WaitVbl or Pause instructions helps assure your program will run at approximately the same speed regardless of whether the user has a 7mc 68000 or a 33mc 68040 CPU. (Everyone learns this the hard way!)

It should be apparent that we could easily substitute "For" loops operating on an object's 'X' and 'Y' registers for the 'Move' command. Because of the automatic Pause done at the end of each iteration, the former scheme is usually superior. As a matter of fact, when OLE is run on an accelerated machine, the bulls' running speeds can be inconsistent. This happens to add a little 'zest' to the game play. For this reason, the 'Move' command has been left in there.

Back to our algorithm: When jumping at F:, we jump instantly 25 lines, but we 'float' at the top for the duration of "RQ" VBlanks. Then, we descend one line per iteration, checking for collisions all the while.

Important!

A reminder that while in AMAL, the 'If' statement brooks no 'Else'. Each test is a "soloist," *and success must result only in a "Jump" instruction!*

Pay close attention to the fact that the matador channel has its very own X, Y, and A variables which are hard coded by AMAL with horizontal and vertical coordinates, and the Sprite Bank image number, respectively. This provides enormous power with few lines of code!

A Bug!!

I have been able to document one bug in AMAL to date. Usually, there is a workaround. The programmer seems to be limited to three "Jump" instructions within some undocumented length of code. Failing to abide this renders an absurd error message. This explains the Mickey Mouse(tm) algorithm used at label C:. I had to figure a way to eliminate one of my Jump instructions, and it was done with the aid of a trick described in the AMOS manual.

C: Let R6=R7>6; Let R9=R7 <-6; If R6|R9 J A:

"R7 > 6" must evaluate TRUE or FALSE. R6 is therefore assigned either -1 or 0. "R7 <-6" must evaluate in like manner, and R9 is assigned -1 or 0. Then, if *either* R6 OR R9 is *true*, the matador is not reasonably centered with the ladder, and we "Jump" back to the start of the program. Previously, I tried to code this with a pair of "If" statements, each of which required a "J A" putting me one over the "Mystic Trinity Finity."

Let's leave the subject of AMAL with a request that you take special note of just how few lines of actual code are required to do so much! The commands are indeed finicky, but there are very few of them for you to master. The results are well worth being 'deviled' a few evenings.

BASIC

Lastly, we reach the BASIC code itself. This is very simple, since the "brain work" is largely contained in the AMAL programs. Study the comments carefully. They are not in proper BASIC format, because I have used them in lieu of writing out notes in standard paragraph form. It is far easier to read explanations when immediately adjacent to the code under discussion.

Besides, it is not intended that you slavishly type this code into your AMOS Editor. AC's editor will make this complete file available to the reader in the best possible way. You may also expect to see a compiled version on Portal et al sometime in the future. Let's have a gander at Table 6.

That's about all there is to it! All these files, stripped of their comments, and despite the inefficiencies of my programming techniques, still total only 7626 code characters! Also, in case you were wondering: OLE's background music is a SoundTracker module borrowed from a Portal library and reformatted with the SoundTracker2_1.AMOS accessory. The command "Music 1" starts the playback and Music Off stops it.

The SampleBankMaker.AMOS accessory converted all the RAW digital samples (as playable by "Sound" from GRAMMA SOFTWARE™) to the form required by the SAM PLAY command. The Sprite Editor allows easy conversion of IFF icons or brushes into

Sprites and Bobs should you prefer using a more potent paint program to the somewhat restricted editor. You will find that, thanks to the AMOS language and several of the many PD utility programs out there, your imagination is the only limit to what you can quickly produce on your Amiga. As far as hours of enjoyment are concerned, I can think of no greater value for so few of your dollars than AMOS. True, C will permit creating programs with faster searches, sorts, and associated grunt work, and a C compiler is another great "bang for the buck." But if visual and audio thrills are important to the task at hand, AMOS is the way to go. AMOS is also simple enough that you will be up and running in a very short time. As a bonus, you don't need an accelerated machine with multi-megabytes of memory to fully exploit it.

For you experimenter types out there, AMOS provides a number of ultra low level commands that permit direct access to the CPU registers out of BASIC! We will save them for another day!

Table Six

```
Table 6
-------
'
'  OLE.AMOS by Thomas J. Engelman  (TOMES@)
'
'     Reading, Pa.  October 1992
'
'        Design by John Gilmore
'
' Note: To improve readability, the comments are not
'       BASIC legal.
'
Load Iff "Ole/Title.pic",2
Load Iff "Ole/Die.pic",5
Load "Ole/OleSprites.abk"  ; Creates and loads bank 1.
Load "Ole/OleAmal.abk"     ; Ditto bank 4.
Load "Ole/OleSamples.abk"  ; Ditto bank 3.
Load "Ole/OleMusic.abk"    ; Ditto bank 1.
'
Screen 2              ; Makes this current and visible.
Music 3               ; Scores title music.
Auto View Off         ; Prevent further automatic display-
ing.
'
Screen Open 1,320,200,16,Lowres  ; Allocates screen for
 Screen Copy 0 To 1              ; second background
pic.
'
Get Sprite Palette : Flash Off : Hide : Double Buffer
'
Screen 0    ; Makes this screen current
Fade 1      ; so we can quickly fade it to black.
Screen 1    ; Makes this current for future operations.
Synchro Off ; Utilized to detect collisions via AMAL.
'
Dim LAD(3)          ; Set horizontal coords for 3 ladders
Dim PRIZE(4)        ; nor horizontal coords for 4 prizes.
'
' All bobs to be assigned an AMAL channel must first be
' declared.  We don't care where they are drawn at this
' time.  Since Auto View is OFF, the draws are actually
' being done in the "invisible" current screen 1.
'
' Bob 0 is the marador.
' Bobs 1,2,3,4 are the bulls.
'             from the bottom deck to top.
' Bobs 5,6,7,8 are the prizes.
'             from the bottom deck to top.
'             See Proc SETPRIZES
'
Bob 0,250,250,5
Bob 1,15,230,0 : Bob 2,104,248,0 : Bob 3,25,230,2
Bob 4,304,246,2 : Bob 5,250,250,10 : Bob 6,260,260,10
Bob 7,270,270,10 : Bob 8,280,280,10
'
' Bobs 9,10,11 are the ladders, bottom deck to top.
' See Proc SETLADDERS
' Bobs 12,13,14,15 are the remaining Marador's icons.
' See Proc SETICONS
'
' Makes the mandatory AMAL channel assignments.
'
Channel 0 To Bob 0 : Channel 1 To Bob 1
Channel 2 To Bob 2 : Channel 3 To Bob 3
Channel 4 To Bob 4 : Channel 5 To Bob 5
Channel 6 To Bob 6 : Channel 7 To Bob 7
Channel 8 To Bob 8
'
' Purpose of Global AMAL Variables
' read or written from BASIC
'
' RA:BULL SPEED.  RC=Skill level (0-8)
' RD=Operations deck (0-3).  RE, RF, RG are ladder X's.
' RH, RI, RJ, RK are prize X's.  RL=Got prize flag.
' RM=Score.  RN=MATADOR SPEED.  RO=Cursed flag.
' RP=4th level done flag  RQ=Jumping delay time.
' RR=Score increase.
'
' Sprite Bank Image Nos. for prizes at each of 9 levels
' For information only.
'
' BAG=10 : BALLOON=11 : SHADES=12 : CONE=13 : CANNON=14
' REFR=15 : LOCO=16 : HOTDOG=17 : WHISKEY=18
'
' Sound File Sample Bank Numbers     Used)
'
BOING=1 : CROWD=2 : CLASS=3 : BITCH=4 : HORN=5
LAUGH=6 : OK=7 : PRIZE=8 : YELL=9
'
'
'  Following is the "main()" program!
'  The next instructions are given once and done when
'   the  game begins.
'
Ink 4,5         ; Writes Blue over White text on screen
1.
Text 2,9,"Ole! Amos 1.0"
'
'   The AMAL code is thus assigned to the AMAL channels.
'
Amal 0,0 : Amal 5,5 : Amal 6,6 : Amal 7,7 : Amal 9,9
Amal On
'
Proc SETICONS       ; Puts little faces in the title bar
                    ; of screen 7.
Volume 64           ; Loud sound samples.
'
   ; Resets AMAL register RD, current ladder X.
Amreg(Asc("D")-65)=0
   ; Resets AMAL reg. RL, the "got prize" flag
```

```
Amreg(Asc("B")-65)=0
  ; Resets AMAL reg. BB, the scoreboard
Amreg(Asc("M")-65)=0
  ; Resets AMAL reg. MO, the 'gored' flag.
Amreg(Asc("O")-65)=0
  ; Resets AMAL reg. OO, the 'scored' flag.
Bkreg(Asc("R")-65)=0
'
Text 27*8,3,"Score 0" ; Writes to 'invisible' screen 1
'
A=5                          ; User gets 5 lives to
lose.
'
' Bullchannels can be assigned only to presently
existing
' bobs. Ergo, draw the bull, priced and man 'off
screen'
' to avoid premature displays. To be sure our bulls
' always restart from the edges of the display, we turn
' these AMAL channels OFF, restarting them at each 'gored'
' or new level. This means channel execution always
' starts over from the very first line of code where X is
' set at the display edges. The matador is handled the
' same way to avoid reading an incidently buffered
' 'collision' on resets.
'
' Simple screen forces 20 secs of music.
Wait 1000
Music Off          ; Enough already!
'
Auto View On       ; Current screen now visible.
'
' This is the start of the main loop. For each of 9
skill
' levels, 0 to 8, we begin by turning off the Matador and
' Bull's AMAL programs. We put the current screen we had
' been drawing to, to the back, forcing screen 0, the
' blackened screen to the front. After making screen 0
' current, we print some bloody fun text on it, and
' flush it with the sprite color palette. In case you're
' wondering, this would make it visible. We rest it up
' over a period of 80 ticks. Next our voice tells the
' user it's OK to play. Keep in mind we're in BASIC now,
' not in AMAL.
'
For Skill To 8
  Amal Off 0 : Amal Off 1 : Amal Off 2
  Amal Off 3 : Amal Off 4
  Screen To Back
  Screen 0
  Ink 4,1  ; Sets blue over can when writing on screen
0
  Text 50,120,"Press Fire button when ready"
  Fade 5 To 1    ; Fade color into the display
(Pauses)
  Wait 80
  Sam Play 33,0

  Do             ; Wait for user to begin
    Wait Vbl     ; in a manner friendly to multi-
tasking.
    If Fire(1) Then Exit
  Loop
'
' We fade viewer's screen to black over 80 ticks. Turn
' display off while screen 3 is being drawn and until
' after we again flip it front. First, make it
' 'current' so that it receives the graphic commands.
' Flush it with color and delay 30 ticks for that
process
' to complete. Since we turned off some AMAL channels,
' we must reassign them before turning them back on.
' The scorekeeping routine in the AMAL price channels
' needs to know what skill level the user is on for its
' scoring algorithm. This value is sent over to AMAL in
' register 'BC'.
'
  Fade 1 : Wait 80
  Auto View Off        ; Flow here, no visible changes.
  Screen 3
  Fade 2 To 1
  Wait 30
  Amal 0,0 : Amal 1,1 : Amal 2,2 : Amal 3,3 : Amal 4,4
  Amal On 0 : Amal On 1 : Amal On 2
  Amal On 3 : Amal On 4
  Bkreg(Asc("C")-65)=8
'
' Develop a short, sweet, suitable algorithm to increase
' the bulls' speed as the skill levels increase (& lower
' 'reset' time value in register 'BA'). I discovered
' quite by accident that you need add a pause at the top
' of the matador's loop in order that the bull can pass
' under him slowly. As the bull reaches a certain
speed,
' you must remove the pause lest the matador have the
' misfortune to meet him during the majestic descent if
' the bull does a quick turn around. This pause value
is
' kept in register 'BD' as examining channel 0 AMAL code
' will reveal. You will experiment with these values to
' suit your reflexes. I am not particularly fast, ^-)
' In like manner, you want to slow down the horizontal
' speed of the matador as the levels increase so as to
' give the bull more chances to get him.
'
  SPEED=(122-(8*S))
  Amreg(Asc("A")-65)=SPEED : 'Ave. time is 60-48
                           ; in decrements of 8
  If SPEED>100
    Amreg(Asc("D")-65)=SPEED/10
'
  Else Amreg(Asc("D")-65)=1
    End If
'
  MANSPEED=(1-S)/2
  If MANSPEED<2
    MANSPEED=2
  End If
  Amreg(Asc("M")-65)=MANSPEED ' Man slows 9-2 thus:
                              ' 2 3 4 4 3 3 2 2 2
'
' Call the functions that return random horizontal
' positions for the ladders and prizes, draw them onto
' the current screen (3), and transmit the returned
' positions to AMAL routines that use them via global
' registers. Call the function that draws the matador
on
' the center of the game screen. Then bring the current
' screen to the front, sending screen 2 to the back.
' Before we forget it, we make this screen 3 current
' momentarily, so we can quickly fade it to black for
' future flipping before the user's eyes. Finally, we
' again make screen 3 current, and render it visible.
'
```

```
 Proc SETLADDERS
 Proc SETPRIZES[[illegible](B-10)]
 Proc SETMATADOR
'
 Screen To Front
 Screen 0
 Fade 1
'
 Screen 1
 Auto View On
'
' The action begins. With each iteration, we first call
' each of the AMAL channels once, then mark time until
' the next screen blank. When this begins, we check the
' flag to be set by Channel 0 in the event the matador is
' sent flying. If so, we play 3 sound samples while
' waiting long enough for each one to complete before
' moving on. We stop the loop, decrement a life, and
' falsely decrement a skill level. I say 'falsely',
' because it is reincremented with the "NEXT B" statement
' to be encountered. The 'gored' flag, AMAL register
' "RG", must be reset for further action.
'
  Do
    Synchro      : Call all AMAL channels simultaneously
    Wait Vbl     ; Stop here, until screen blanks.
'
    If Amreg(Asc("G")-65)=1   ; then matador was gored.
      Sam Play $F,BOING
      Wait 50
'
      Bob Off A+10
      Dec A : Dec B
      Amreg(Asc("G")-65)=0
      Sam Play $F,YELL
'
      If A=1            ; Then we have no more lives
        Exit
      End If
'
      Fade 15
      Wait 100
      Sam Play $F,CLASH
      Wait 100
      Exit
    End If
'
' If we are not gored, we should next test AMAL register
' RR to see if the user has captured a prize, and make a
' sound and increase the printed score value if so. We do
' this by saving the last score in RR. RM always has any
' new value set by an AMAL prize channel. We need only do
' a fast and snappy "compare" to see if the matador
' grabbed a prize.
                          ; If new Score > old score
    If Amreg(Asc("R")-65)<>Amreg(Asc("M")-65)
      Sam Play $8,BRIGHT
     Text 220,8,"SCORE"+Str$(Amreg(Asc("M")-65))
'                         ; make old score = new score
     Amreg(Asc("R")-65)=Amreg(Asc("M")-65)
    End If
'
' And lastly, we check on the status of AMAL register RP.
' This is set only by the AMAL channel belonging to the
' uppermost prize. If this has been captured by the user,
' he has completed a level. Reset the flag, make joyful
' noises, and fade out the screen. (more code would be
' useful here)
'
    If Amreg(Asc("P")-65)=1    ; top deck's prize hit
      Amreg(Asc("P")-65)=0
       Sam Play $9,HORN
       Sam Play $A,CROWD
        Wait 50
        Fade 15
        Wait 200
        Exit
     End If
'
   Loop               ; Mates with 'Do'
'
   If A=1             ; If we are out of lives (icons)
     B=10             ; fool B into exceeding its bounds,
   End If             ; thus dropping out of the loop.
'
' If, upon dropping out of the B loop we find the value of
' B to have been 8, then the user must have successfully
' completed the game. Reward him with some sound effects,
' a slow fade out, and then quit the program with an END
' statement to avoid the loud guffaws that otherwise ensue
' when leaving the program.

   If B=8
     Sam Play $F,HIGH
      Wait 50
     Sam Play $5,HORN
     Sam Play $A,CROWD
      Fade 15
      Wait 225
       End
   End If
 Next B
'
' If we drop out the B loop for any reason other than that
' B=8, it must be because "Slow Hands" ran out of matador
' lives; ie: he screwed up.  Give him the raspberry
' before fading out and returning to the workbench.
'
Sam Play $A,LAUGH
Wait 50
Sam Play $F,LAUGH
Fade 15
Wait 225
'
 End               ;Return to the WBench.
'
' In this procedure, we develop 3 random locations from 32
' to 308 at which to place the hot spots for the 3 ladders
' (keeping them entirely on the display). Their Y
' coordinates are fixed at 96, 146 and 196. The X
```

```
' coordinates are stored in the LAD array, and loaded
into
' the global registers RE, RF and RG for use by AMAL. We
' use a little trick to assign the lowest ladder first.
' We also make doubly certain that register RD is zero to
' begin with before it is used in the channel 9 routine.
'
Procedure SETLADDERS
   Shared LAD()
   B=196
   For A=0 To 2
     LAD(A)=Rnd(296)+12
     Amreg(A+4)=LAD(A)
    Bob A+9,LAD(A),B-(A*54),1
   Next A
  Amreg(Asc("D")-65)=0
End Proc
'
' We place one prize on each level at a random horizontal
' position similar to the ladder placement algorithm.
' Note this Procedure takes an argument that depends upon
' the skill level. The positions are kept in the PRIZE
' array and are fed into AMAL registers RH, RI, RJ and RK
' for use in the prize channels. We will also make
' certain RL, the 'got prize' flag is reset.
'
Procedure SETPRIZES[LEVEL]
   Shared PRIZE()
   B=185
   For A=0 To 3
     PRIZE(A)=Rnd(296)+16
     Amreg(A+7)=PRIZE(A)
    Bob A+5,PRIZE(A),B-(A*54),LEVEL
   Next A
  Amreg(Asc("L")-65)=0
End Proc
'
' We use this Procedure to place four "life" icons on the
' title bar. We use four separate bobs for this to make
' our life easy when it comes to removing one whenever a
' matador is lost simply by calling the BOB OFF command.
'
Procedure SETICONS
   For A=0 To 3
     Bob A+12,(A*14)+100,2,5
   Next A
End Proc
'
' This Procedure is for completeness only. It does
' nothing but consistently respot the matador bob in the
' exact center of the bottom level when called.
'
Procedure SETMATADOR
   Bob 0,160,196,0
End
```

Listing One

```
' chan 0 to bob 0
' the matador
'
' [illegible]
' [illegible]
' [illegible]
[illegible]
'
'''' at the moment, [illegible]
[illegible]
'
 [illegible]
'
' [illegible]
'
A: [illegible]
B: If [illegible] Jump [illegible]
C: If [illegible] Jump [illegible]
D: If [illegible] Jump [illegible]
E: If [illegible] Jump [illegible]
'
' [illegible]
'
  [illegible]
'
' [illegible]
quickly, pause for a
' [illegible]
check for collisions
' [illegible]
'
F: [illegible]
   [illegible]
[illegible]
'
' [illegible]
'
G: [illegible]
H: [illegible]
'
I: [illegible]
[illegible]
'
' [illegible]
[illegible]
' [illegible]
collide.
'
J: [illegible]
[illegible]
K: [illegible]
[illegible]
'
' [illegible]
pertinent ladder's
'
L: [illegible]
M: [illegible]
N: [illegible]
```

(continued on page 68)

Technical Writers
Hardware Technicians
Programmers
Amiga Enthusiasts

Do you work your Amiga to its limits? Do you do create your own programs and utilities? Are you a master of any of the programming languages available for the Amiga? Do you often find yourself reworking a piece of hardware or software to your own specifications?

If you answered yes to any of those questions, then you belong writing for AC's TECH!

AC's TECH for the Commodore Amiga is the only Amiga-based technical magazine available! We are constantly looking for new authors and fresh ideas to complement the magazine as it grows in a rapidly expanding technical market.

Share your ideas, your knowledge, and your creations with the rest of the Amiga technical community—become an ACs TECH author.

For more information, call or write:

AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

1-800-345-3360

Programming the Amiga in Assembly Language

Part VI - Gadgets, Strings, and Other Things

by William P. Nee

In this article I'll discuss gadgets, creating a string gadget, using the BORDER structure and text, how to scale numbers, and use the Intuition message structure to make a zoom routine. We'll write a Mandelbrot/Julia program to tie all this together.

Menu Macro

But first, some old business. In the last article (V2:4) I discussed menus, items, and sub-items; this code took up a lot of room in the program and I suggested you devise your own macros to cut down on some of the repetitive work. Here's how I wrote my macros—included in the MENU.i listing on this disk.

```
MAKEMENU MACRO                                  ;\1 = this menu (menu0, menu1, etc.)
\1                                              ;\2 = 'menu title'
              IFNC    '\3',''                   ;\3 = next menu or 0
              DC.L    X                         ;\4 = left edge (0,100,200,...)
              ENDC                              ;\5 = flags (1=enabled, 0=disabled)
              IFC     '0','\3'
              DC.L    0
              ENDC
              DC.W    \4,0,90,0,\5
              DC.L    .\1TITLE                  ;title pointer
              DC.L    \1ITEM0                   ;first item structure pointer
              DC.W    0,0,0,0
              EVENIFC
.\1TITLE      DC.B    .2,0                      ;reserve space for title
              EVENIFC
              ENDM
```

The first part of the macro checks to see if there is another menu title; if so, it makes this the first entry and if not, it makes the first entry a NULL. The next entry is the left-edge followed by a top-edge of 0. The width is set to 90 and the height is NULL. Your menu flag is next and the macro then fills in the pointer to the menu title and the first item structure. The four unused words complete the structure. The menu title, your second parameter, completes the macro. This macro would be used as:

```
MAKEMENU MENU0,'Project',MENU1,0,1
```

Notice the \1 and .\1TITLE used to make the headings. This is the menu that the OPENMENU macro looks for.

Item Macro

Next, and a little more complicated, is the item macro. Now we have to pass a few more parameters. Here's one way to write it:

```
MAKEITEM MACRO                                  ;\1 = this item
(menu0item1, etc.)
\1                                              ;\2 = 'item
name'
              IFNC    '\3',''                   ;\3 = next item or 0
              DC.L    \3                        ;\4 = top
(0,10,20,...)
              ENDC                              ;\5 =
flags
              IFC     '\3',''                   ;\6 = mutualexclude
or 0
              DC.L    0                         ;\7 =
"command key" or 0
              ENDC                              ;\8 =
menu_item_subitem_pointer
              DC.W    0,\4,130+CHECKWIDTH,10,\5
              IFNC    '\6',''                   ;\6 => nothing
              DC.L    \6
              ENDC
              IFC     '\6',''                   ;\6 = nothing
              DC.L    0
              ENDC
              DC.L    .\1ITEXT                  ;IText structure
pointer
              IFNC    '\7',''                   ;\7 => nothing
              DC.B    \7,0                      ;command
key, padding
              ENDC
              IFC     '\7',''                   ;\7 = nothing
              DC.B    0,0
              ENDC
              IFNC    '\8',''                   ;\8 => nothing
              DC.L    \8                        ;pointer to
subitem structure
              ENDC
              IFC     '\8',''                   ;\8 = nothing
              DC.L    0
              ENDC
              DC.W    0
.\1ITEXT
              DC.B    0,1,1,0                   ;front pen, back pen,
mode, padding
              DC.W    CHECKWIDTH,0              ;width offset
              DC.L    0,.\1NAME,0
              EVENIFC
.\1NAME       DC.B    .2,0                      ;reserve space for
item name
              EVENIFC
              ENDM
```

As with the menu macro, either the next item or NULL becomes the first macro variable. This is followed by a left-edge of 0, your top location, a width of 130 plus room for the command key and symbol, and a height of 10. Your flags for this item are next, followed by the mutual exclude, if any, the pointer to the IText structure, your command key, if any, and the pointer to any subitem structure. A next select of 0 finishes this part. The IText portion includes a left-edge offset of CHECKWIDTH to always allow for the check mark. The item

name completes the macro. An example of this macro would be:

```
MAKEITEM MENUITEM0,'ItCheat',,0,$52,,MENUITEM0SUBITEM0
SUBITEM MACRO
```

The macro for creating subitems is very much like the item macro and passes all but the last parameter.

```
MAKESUBITEM MACRO                       ;\1 = this subitem
;MenuItem0subitem0
\1                                                      ;\2 =
'subitem name'
        IFNC    '\3',''                 ;\3 = next subitem or
0
        DC.L    \3                                      ;14 = top
(0,10,20...)
        ENDC                                            ;15 =
flags
        IFC     '\5',''                 ;16 = $mutualexclude
or 0
        DC.L    0                                       ;17 =
'command key'
        ENDC
        DC.W    65,\4,$30+COMMODR,10,\5
        IFNC    '\6',''                 ;\6 = nothing
        DC.L    \6                                      ;mutual
exclude
        ENDC
        IFC     '\6',''                 ;\6 = nothing
        DC.L    0
        ENDC
        DC.L    \1TEXT,0                ;IText structure
pointer
        IFNC    '\7',''
        DC.B    \7,0                                    ;command
key, padding
        ENDC
        IFC     '\2',''
        DC.B    0,0
        ENDC
        DC.L    0                                       ;no sub-
item structure
        DC.W    0
        EVENPC
\1ITEXT
        DC.B    0,1,1,0                 ;front pen, back pen,
mode, padding
        DC.W    CHECKWIDTH,0            ;width offset
        DC.L    0,\1NAME,0
        EVENPC
\1NAME  DC.B    (),0                    ;reserve space for
name
        EVENPC
        ENDM
```

Notice that I have subitems starting 65 spaces out from the menu strip and they include room for the check mark and command key. An example of the subitem macro would be:

```
MAKESUBITEM MENUITEM0SUBITEM0,'$4',MENUITEM0SUBITEM1,0,$153,$1E
```

These three macros are "generic" in that they all produce the same size and style menu, item, and subitem. You could REM portions or change parts of any macro, but be sure to keep the original intact. If you've got your own macros, by all means use them, and adjust the program accordingly.

Gadgets

Now let's see how you can communicate with your program by using gadgets. Intuition provides several System gadgets—closing, front/back, drag, etc—but you can also custom-design your own. In general, there are three types of gadgets:

1) Boolian ($1) - ON/OFF button
2) Proportional ($3) - a sliding knob inside a container
3) String ($4) - allows entry of a string or a long integer number

These gadgets can be highlighted just like menus and can have text associated with them. You may design your own images for the regular and highlighted gadget or enclose them in a border. I'll save image-making for a later article, but we will use a border.

Borders

The border function of Intuition graphics will draw lines between any given sets of coordinates relative to the container they're inside. The border structure is in the second half of Table II. The left-edge and the top-edge are offsets from the container box so they're usually negative numbers. The draw mode is either JAM1 or COMPLIMENT/XOR. Let your gadget know there will be a border by including the border structure pointer in the gadget structure. In addition to attaching a border structure to your gadget you can draw lines anytime using the DrawBorder function. Set up this function with:

A0 = RASTPORT
A1 = BORDER STRUCTURE POINTER
D0 = X COORDINATE OFFSET
D1 = Y COORDINATE OFFSET

You could draw the same lines at several different places on the screen by changing d0 and d1.

Gadgets are structured items as outlined in Table I. As with most major structures, the first entry is a pointer to the next gadget's structure. The locations for the gadget are next followed by the various flag options. Flags for gadgets are:

```
GADGHCOMP ($0) - complement the rutnor; used for string gadgets
GADGHBOX ($1) - draw a box around the gadget
GADGHIMAGE ($2) - a highlighted image or border
GADGHNONE ($3) - no highlighting
GADGIMAGE ($4) - user defined image
GRELBOTTOM ($8) - top-edge is relative to bottom
GRELRIGHT ($10) - left-edge is relative to right-edge
GRELWIDTH ($20) - width is relative to window width
GRELHEIGHT ($40) - height is relative to window height
SELECTED ($80) - starts on and highlighted if toggled
GADGDISABLED ($100) - starts off and disabled
```

To better understand the GREL options, if you set the height to 9 the gadget will be nine lines high; but set height to -50 along with GRELHEIGHT and the gadget will be 50 lines smaller than its element, the window. A width of -100 combined with GRELWIDTH will produce a gadget 100 pixels smaller than the window width. In the same manner, GRELRIGHT has the starting point relative to the right side of the window and GRELBOTTOM has the starting line relative to the window bottom. These flags let you place a gadget inside any size window using any screen mode (320 X 200, 640 X 200, etc.).

The Activation flags produce desired effects and further define the gadget. Activation flags are:

```
RELVERIFY ($1) - active when LMB is released over the gadget
GADGIMMEDIATE ($2) - know immediately when gadget selected
ENDGADGET ($4) - make a requester gadget go away
FOLLOWMOUSE ($8) - follows the mouse; for proportional gadgets
RIGHTBORDER ($10) - adjust the window border
LEFTBORDER ($20) - adjust the window border
TOPBORDER ($40) - adjust the window border
BOTTOMBORDER ($80) - adjust the window border
TOGGLESELECT ($100) - toggle ON-OFF states
STRINGCENTER ($200) - center justify a string entry
STRINGRIGHT ($400) - right justify a string entry
LONGINT ($800) - string must be a long integer
```

```
ALTKEYMAP ($1000) - there will be an alternate keymap
BOOLEXTEND ($2000) - there is a Boolean info structure
```

The BORDER options may be used to change the size of a window's border so you can place your gadget there. TOGGLESELECT is used in conjunction with the SELECTED flag. STRINGCENTER/RIGHT will position the placement of your string information as you enter it in the gadget box—like the RENAME function in WorkBench. LONGINT lets Intuition know your string will only be a long integer (32-bits, signed).

Next in the structure is the gadget type. This is followed by three pointers to an image/border structure, highlighted image/border structure, and an IntuiText structure. After the mutualexclude is a pointer to further information for proportional or string gadgets. You may include a gadget number to define which gadget is in use as well as your own structure pointer at the end of the gadget structure. Intuition will disregard these last two entries.

Various flags, activation flags, and structure elements must be combined to produce the desired results. The major combinations are:

```
FLAGS                   ACTIVATION                      STRUCTURE
SELECTED ($80)          TOGGLESELECT ($100)   MUTUALEXCLUDE
IMAGEIMAGE ($4)                                                 IMAGE/
BORDER POINTER
IMAGEIMAGE ($2)                                                 H.IMAGE/
H.BORDER POINTER
```

For this listing all the gadgets will use their own shape and be in their regular locations, so I used a flag of $0 and, since I wanted to know only when the LMB was released, I used an activation flag of $1.

STRINGINFO

If your gadget is a string or proportional type you must also include an info structure. Since I'll only be using string gadgets, I'll discuss some of the items in the stringinfo structure outlined in Table II. Each string must have a buffer where the string will be stored and this buffer must be NULL terminated. You may also have an optional undobuffer where the last string entry can be recalled using AMIGA/Q. Since only one string gadget is active at a time, multiple gadgets can share the undobuffer. In this article, though, I'll have separate undobuffers for each string gadget.

The maximum numbers of characters allowed in the buffer must also include that NULL terminating zero. Intuition maintains the next several fields but you can look up their values as an offset from the stringinfo structure. If you're going to have a long integer string gadget, you must not only set the LONGINT ($800) activation flag, but must also include the initial long integer in the stringinfo structure. As this integer changes, you can look up its value. If you have an alternate keymap, set the ALTKEYMAP ($1000) activation flag and include its location as the last item in the stringinfo structure.

STRGADGET MACRO

As you can see, it takes a lot of code to define several string gadgets. Listing 1 will use six gadgets and would require 168 additional lines of program code to describe all of them. Again, we'll use macros that will shorten the work for us. Since there are so many string possibilities, we'll have to pass 12 items, but if you want your macro to be more generic you could cut down on some of them. My string gadget macro follows the format of the menuitem macro:

```
MAKESTRGADGET MACRO
\1
\1 = this string gadget
        IFNC    '\3',''                                 ;\3 =
'string gadget name'
        DC.L    \3
\3 = next string gadget
        ENDC
\4 = left-edge offset
        IFC     '\3',''                                 ;(5 = top-
edge offset
        DC.L    0
\6 = width
        ENDC
;\7 = height
        DC.W    \4,\5,\6,\7                             ;\8 = flags
        DC.W    \8,\9,4                                 ;\9 =
activation
        DC.L    \10+256,0                               ;\10 = &mutualexclude
or 0
        DC.L    \11TEXT,\12                             ;\11 = id# or 0
        DC.L    \13INFO                                 ;\12 = max
# of characters
        DC.W    \14
        DC.L    0
        EVENPC
\13INFO
        DC.L    \14BUFFER,\15UNDOBUFFER  ;buffer, undobuffer pointers
        DC.W    0,\12,0,0,0,0,0,0
        DC.L    0,0,0
        EVENPC
\10BORDER
        DC.W    -2,-2                                   ;start-log
offsets
        DC.B    1,0,0,5
        DC.L    \POINTS,0                               ;pointer to coords;
nates
        EVENPC
\POINTS
        DC.W    0,0,\6+4,0,\6+4,\7+4,0,\7+4,0,0
        EVENPC
\11TEXT
        DC.B    0,1,1,0
        DC.W    0,-11
        DC.L    0,\11NAME,0
        EVENPC
\11NAME DC.B    \2,0                                    ;reserve
space for name
        EVENPC
\14BUFFER
        DS.B    \12-1                                   ;reserve
buffer space
        DC.B    0
        EVENPC
\15UNDOBUFFER
        DS.B    \12-1                                   ;reserve
undobuffer space
        DC.B    0
        EVENPC
        ENDM
```

Let's look at this macro in a little more detail. The first item is either the pointer to another string gadget or 0. This is followed by the locations for your gadget, flags, activation flags, and the type gadget (4). Each gadget has a border pointer, no highlighting pointer, an IntuiText pointer, no mutualexclude, and a stringinfo pointer. Your id# follows and there is no pointer to an optional structure.

The stringinfo portion has both a buffer and undobuffer pointer. Characters will start printing at position 0 in the gadget box and you define how many characters there may be. The border offsets are two pixels to the left and two lines above the gadget container. There are five pairs of coordinates to get back to the origin. The five pairs are (0,0), (your width+4,0), (your width+4, your height+4), (0, your height+4), and (0,0). The IntuiText starts 11 lines above the container. Space is reserved for each buffer and is one byte less than you want plus the NULL terminating 0.

The easiest way to include your gadgets is to add the pointer to the first gadget to your window structure, offset 18. You can remove the gadgets by replacing the pointer with a NULL; this is what the

REMOVEGADGETS macro does. All of these macros and the appropriate flag values have been added to MENU.i included on this disk.

The Program

Now let's put all of this knowledge to use in a program. In the last article I talked about the Julia Set and showed you how to draw it using double-precision math. This time we'll draw the Mandelbrot Set using scaled numbers. There will be a string gadget for each of the variables—Xleft, Xright, Ybottom, Ytop, JuliaA, and JuliaB; you can put whatever values you want in these strings. Menu selections will let you pick a program to draw either Mandelbrot or Julia, go back to the coordinates, or quit the program. Another menu selection will let you pick a maximum iteration count from 64 (the default count) to 1024. During any part of the drawing you can use the LMB to pick the upper-left corner of a zoom box, drag down to the lower-right corner, release the button and recompute new coordinates to draw. The original values in the string gadgets will not change. I know there are quicker ways to draw the Mandelbrot Set, but this is a program designed to show you how to use menus, gadgets, and IntuiMessage.

To review, the Julia Set was derived by continuously squaring a complex number and adding a fixed number to the result. Keep repeating this until the value of the number exceeds 4, or you reach the maximum iteration count. Numbers that never exceed 4 are part of the Julia Set and usually colored black. Numbers that exceed 4 are usually colored based on the iteration count at that point. This process is repeated for every complex number within a grid. Remember that a complex number Z is represented as X+iY where i is the square root of -1, so i*i=-1. The real part of a complex number is the X portion and the imaginary part is the Y portion. The complex number Z squared is (X+iY)*(X+iY) or X^2+2iXY+i^2Y^2. Since i^2=-1, new Z is X^2-Y^2+i2XY; the new real portion is (X^2-Y^2) and the new imaginary portion is 2XY. Then JuliaA and JuliaB are added to the real portion and imaginary portion respectively.

The only difference between the Julia and Mandelbrot computations is the number added to the new real and imaginary portions. Constant Julia values are added to the Julia Set; continuously varying values, the Xcorner and Ycorner are added to the Mandelbrot Set. Here are the two programs in BASIC to show you the difference.

```
 JULIA SET
XLEFT=-2:XRIGHT=2:XSCALE=(XRIGHT-XLEFT)/64
YBOTTOM=-2:YTOP=2:YSCALE=(YTOP-YBOTTOM)/64
JULIAA=-1.5:JULIAB=0:MAXCOUNT=64
FOR H=0 TO 64:XCORNER=XLEFT+H*XSCALE
FOR V=0 TO 64:YCORNER=YBOTTOM+V*YSCALE
A=XCORNER:B=YCORNER
FOR C=0 TO MAXCOUNT:ASQR=A*A:BSQR=B*B
IF ASQR+BSQR>=4 THEN COLOR ROUTINE:GOTO LOOP
B=2*A*B+JULIAB:A=ASQR-BSQR+JULIAA
NEXT C
LOOP:NEXT V,H
MANDELBROT SET
XLEFT=-2:XRIGHT=2:XSCALE=(XRIGHT-XLEFT)/64
YBOTTOM=-2:YTOP=2:YSCALE=(YTOP-YBOTTOM)/64
MAXCOUNT=64
FOR H=0 TO 64:XCORNER=XLEFT+H*XSCALE
FOR V=0 TO 64:YCORNER=YBOTTOM+V*YSCALE
A=XCORNER:B=YCORNER
FOR C=0 TO MAXCOUNT:ASQR=A*A:BSQR=B*B
IF ASQR+BSQR>=4 THEN COLOR ROUTINE:GOTO LOOP
B=2*A*B+YCORNER:A=ASQR-BSQR+XCORNER
NEXT C
LOOP:NEXT V,H
```

Two different ways to color a point are to AND the iteration count with #31 or to shift it enough to the right to get a value from 0 to 31. The first method may produce a jumble of colors where there should be a pattern.

Scaling Numbers

When we computed the Julia Set I used double-precision floating-point numbers, but this time we'll scale each number by multiplying it by a large factor and then use the registers for regular multiplication rather than the slower MATHIEEEDOUBBAS dp functions. The scale in this program is 2^29. The first step is to convert the string to a dp number just as in the Julia program, but then use the MSCALE macro to multiply it by 2^29 (as a dp number) and then move it back to d0 as a whole number. This is done for each string value and the scaled numbers are stored in XC, YC, JULIAA, and JULIAB. The difference between Xright and Xleft is divided by 320 and stored in XSCALE; the difference between Ytop and Ybottom is divided by 200 and stored in YSCALE.

If you pick the menu option MANDELBROT, the Julia flag is set to 0; if you pick the JULIA option, it's set to 1. The CFM macro does not use Wait since we want the program to keep drawing unless a menu item or the zoom routine is picked. We also need the mouseX and mouseY coordinates so I rewrote the CFM macro as follows:

```
CFM MACRO                                       ;<branch to if no message>
            MOVEA.L     WINDOW,A0
            MOVE.L      #0,D0
            CMTLIB      GETMSG
            MOVEA.L     WINDOW,A0
            MOVEA.L     MW_USERPORT(A0),A0
            SYSLIB      GETMSG
            TST.L       D0
            BEQ                 \1
            MOVEA.L     D0,A1
            MOVE.L      IM_CLASS(A1),D2         ;IDCMP flags
            MOVE.W      IM_CODE(A1),D3                  ;mouse#, LMB/RMB up/
down, etc.
            MOVE.W      IM_QUALIFIER(A1),D4     ;rawkey code
            MOVEA.L     IM_IADDRESS(A1),A0      ;address
            MOVE.W      IM_MOUSEX(A1),D5        ;X coordinate
            MOVE.W      IM_MOUSEY(A1),D6        ;Y coordinate
            SYSLIB      REPLYMSG
            ENDM
```

Now let's go through Listing 1 in detail and review all of the subroutines. There are six include files necessary to run this program. Even though we're going to scale numbers, I still used DPMATHMACROS.i for division and to move multiple registers around. MULR is my macro for multiplying the unsigned values in d0 and d1 with the result in d2/d3. It uses shifts and rotate commands rather than the MULU function. ZMUL will be used to multiply two labels with the result in d0. The current iteration count is stored in register a2 and your maximum iteration count is in a3. Using registers rather than labels to store these frequently called values will speed up the program a little.

@PSET is a variation of the PSET macro so I put the @ sign before it to keep the assembler from becoming confused. MSCALE will multiply a dp value in d0/d1 by 2^29, return the result in d0, and save it in the passed location. BEEP will flash the screen when called, acting as sort of a minor alert. More about using it later. After the libraries are opened the screen and window are set up along with an initial maximum iteration count of 64.

When you first see the program there will be six string gadgets on the screen filled with default Mandelbrot values. You may use these values or click in any box with the LMB and change the values or you can use AMIGA/X to erase the block; use AMIGA/Q to restore the last entered value. When you've changed values, press <ENTER> and then

change whatever other values you want. The Mandelbrot program does not use JuliaA and JuliaB; the Julia program display is generally between -1.5 and 1.5 in both directions unless you want to zoom in on a portion of it.

The message check sits there patiently waiting to see if you've picked a menu option. I have it set to only react to the Project menu. When you choose either Mandelbrot or Julia the respective flag is set and the program starts to compute the scaled coordinates. The first step is to remove the existing gadgets otherwise you could still click inside the invisible box and get a cursor along with the current string value—this does not make for an interesting display. The string value in each of the six buffers is converted to a dp number using the CONVERTDP macro I discussed in the previous article. The dp value is then scaled by multiplying it by 2^29; the scaled Xleft value is stored in XC. When the new Xright value is computed it's not saved since it doesn't enter into the calculations, but the previously saved Xleft is subtracted from it. This difference is divided by 320, scaled, and saved as XSCALE. The same procedure is followed with Ybottom except that the difference between Ytop and Ybottom is divided by 200 before scaling it. Finally the JuliaA and JuliaB values are always scaled and saved.

Since they will keep changing, the original XC and YC are resaved as XLOC and YLOC; for each loop they are also saved as ALOC and BLOC. The current count and sign flag are both set to 0. Now check the value in ALOC for it's sign, if it's negative, negate it and add 1 to the sign flag. Then use MULR to square the value. Remember that this value is actually ALOC*ALOC*SCALE*SCALE. Repeat the same procedure with BLOC and then add ASQR+BSQR. ADDX.L will include any carry from d1+d3 when you add d0 and d2. Compare the first half of the number (in d2) to #$10000000 which is the left half of 4*SCALE*SCALE. If it's less, we'll go on to compute the new imaginary portion of our number. If it isn't less, use one of the optional coloring programs to fix the color, set the point, and then branch to FIN.

New Z

The new imaginary portion is 2*ALOC*BLOC, but again, since each value is scaled, just multiplying would produce a new value of 2*ALOC*SCALE*BLOC*SCALE. So we'll have to multiply then divide our answer by SCALE to get a single scaled value. Rather than multiply by two and then divide by 2^29 we'll just divide by 2^28, eliminating one step. Division is accomplished using 28 right shifts; ROXR.L will include any carry from the ASR.L of d2. If the sign flag contains a one the value in d2/d3 is negative. Finally, if the Julia flag is set, JuliaB is added else YLOC is added, resulting in the new imaginary portion.

Now the new real portion must be computed. Its value is ALOC*ALOC-BLOC*BLOC. But again, since each value is scaled the final answer will be too large and must be divided by the scale. We already have the values saved for ASQR and BSQR so subtract them and divide by the scale using 29 right shifts. Add either JuliaA or XLOC and there's the new real portion of our new complex number.

Increase count by one and compare it to the maximum count—both values are actually in address registers. If the current count is below the maximum, branch to AGAIN. If the maximum count is reached the location is inside one of the two sets, so either leave it the background color or set it to the color of your choice. Then add the YSCALE to YLOC to get the next coordinate.

At this point we need to check for any messages; you could decide to pick menu0, menu1, or begin the zoom routine. If the CFM macro does not receive a message the program branches to NO_MESSAGE where the down distance is increased by one and if we're not at the bottom of the screen the program goes back to ML2. If there is a message however, the program must check to see it it's a MENUPICK or a MOUSEBUTTON; if it's neither, then again branch to NO_MESSAGE.

If there is a MENUPICK then the menu, item, and subitem number are computed. In menu0 you could select Mandelbrot in which case the julia flag would be cleared and the program jumps to START. Likewise, if you choose Julia the julia flag is set and the program jumps to start. If you pick Coordinates the program clears the screen, closes the menustrip and window, then jumps to the MAKE_WINDOW routine where the window is reopened along with it's gadgets. Finally, you could opt for Quit in which case the program would branch to CLOSE_WINDOW and end the program. If you don't choose any of these items the program branches to NO_MESSAGE. You can select menu items by using either the LMB or the item's corresponding command key.

Or you might have chosen menu1 and one of its five subitems. Depending on which subitem is selected, the new maximum iteration count is stored in MAXCOUNT (register a3). Since no further action is needed the program immediately jumps to NO_MESSAGE after you change the iteration count.

ZOOM

If you activate the MOUSEBUTTON flag by pressing the LMB the program branches to ZOOM. The values in MouseX and MouseY are made into words, stored in STARTX and STARTY, and the draw mode switched to complement. With this mode a line drawn over itself restores the original pixel values under it. Another message check is made to see if you move the mouse or release the LMB. When you do release the LMB, the IM.CODE in d3 will equal SELECTUP (#$E8). If you haven't released the button, the coordinates in MouseX and MouseY are now stored in ENDX and ENDY. The BOX macro will draw a box connecting STARTX, STARTY, ENDX, and ENDY. After an optional delay the same box is redrawn restoring the original color of the pixels under the line. The program then branches back to the message check.

When the message check says there's a MOUSEBUTTON and the code says it's SELECTUP, the program branches to LMB_UP. There the original mode of JAM1 is restored, then ZMUL is used to multiply XSCALE*STARTX. The result is added to XC and saved as NEWXC; this is already a scaled number. Then multiply XSCALE times ENDX, add it to XC, and subtract the just computed NEWXC to get the distance between your X points. Convert this to a dp number, divide by 320, return it as a whole number in d0, and save it as the new XSCALE. Before saving it, however, check to be sure the scale is at least 1, otherwise you've exceeded the program's maximum zoom capability. If it is a 0, change it to one and BEEP the screen to let the user know there's no more zooming.

The same procedure is repeated with YSCALE, but, since the bottom of the screen is actually line 200 and the top is actually 0, you must reverse everything. Subtract ENDY from 200 then multiply by YSCALE, add YC, and save the result as NEWYC. Subtract STARTY from 200, multiply by YSCALE, add YC, subtract NEWYC, and divide by 200. Again test this number to be sure the new YSCALE is not 0 and

BEEP if it is. When all of the new locations and scales have been computed, branch to SHOWIT. These new locations do not affect the values in the string buffers, nor do you know what they are.

The final routine adds the XSCALE to XLOC, increases the across distance by 1, and, if we're not done, branches to ML1. If the drawing is complete the program branches back to the message check to see what you want to do.

The variables XC, YC, XSCALE, and YSCALE have been computed for the default values of -2, 2, -2, and 2. Because I wanted these values to appear in the string gadgets I REM'd the buffer portion of MAKESTRGADGET macro and used buffers at the end of the listing filled with the desired values; notice that each one must be NULL terminated. Assemble this program using A68K as M\J, or copy it from the enclosed disk.

I hope this article has helped you to understand IDCMP flags, IntuiMessages, and how they work together with menus and gadgets. Some changes you might want to make to this program are to disable portions of the menu that aren't in use, react to menu1 at the start of the program, and to replace new coordinates in their respective string buffers. Experiment with the string gadget placement, menu colors, and text colors. Most of all, keep working with assembly language; the more you use it, the easier it becomes. And let those macros do a lot of the work for you.

TABLE I

TABLE I
GADGET STRUCTURE (44 BYTES)

0	LONG		POINTER TO NEXT GADGET STRUCTURE		
4	WORD		LEFT-EDGE RELATIVE TO WINDOW		
6	WORD		TOP-EDGE RELATIVE TO WINDOW		
8	WORD		WIDTH OF GADGET BOX		
10	WORD		HEIGHT OF GADGET BOX		
12	WORD		FLAGS		
		GADGHCOMP ($0)	GADGHBOX ($1)	GADGHIMAGE ($2)	GADGHNONE ($3)
		GADGIMAGE ($4)	GRELBOTTOM ($8)	GRELRIGHT ($10)	GRELWIDTH ($20)
		GRELHEIGHT ($40)	SELECTED ($80)	GADGDISABLED ($100)	
14	WORD		ACTIVATION		
		RELVERIFY ($1)	GADGIMMEDIATE ($2)	ENDGADGET ($4)	FOLLOWMOUSE ($8)
		RIGHTBORDER ($10)	LEFTBORDER ($20)	TOPBORDER ($40)	
		BOTTOMBORDER ($80)	TOGGLESELECT ($100)	STRINGCENTER ($200)	STRINGRIGHT ($400)
		LONGINT ($800)	ALTKEYMAP ($1000)	BOOLEXTEND ($2000)	
16	WORD		GADGET TYPE		
		BOOLGADGET ($1)	PROPGADGET ($2)		
		STRGADGET ($4)	REQGADGET ($1000)	GZZGADGET ($2000)	
18	LONG		POINTER TO IMAGE OR BORDER STRUCTURE		
22	LONG		POINTER TO HIGHLIGHTED IMAGE OR BORDER STRUCTURE		
26	LONG		POINTER TO GADGET'S INTUITEXT STRUCTURE		
30	LONG		MUTUALEXCLUDE		
34	LONG		POINTER TO PROPINFO OR STRINGINFO STRUCTURE		
38	WORD		USER-DEFINED GADGET NUMBER		
40	LONG		POINTER TO USER-DEFINED STRUCTURE		

TABLE II

STRINGINFO STRUCTURE (36 BYTES)

0	LONG	POINTER TO STRING BUFFER
4	LONG	POINTER TO STRING UNDOBUFFER
8	WORD	CURSOR LOCATION IN THE BUFFER
10	WORD	MAXIMUM NUMBER OF CHARACTERS IN THE BUFFER
12	WORD	FIRST CHARACTER LOCATION IN THE BUFFER
14	WORD	CHARACTER POSITION IN THE UNDOBUFFER - INT SET
16	WORD	NUMBER OF CHARACTERS IN THE BUFFER - INT SET
18	WORD	NUMBER OF VISIBLE CHARACTERS - INT SET
20	WORD	LEFT-OFFSET OF CONTAINER - INT SET
22	WORD	TOP-OFFSET OF CONTAINER - INT SET
24	LONG	POINTER TO LAYER STRUCTURE - INT SET
28	LONG	VALUE OF THE LONG INTEGER - INT SET
32	LONG	POINTER TO AN ALTERNATE KEYMAP

BORDER STRUCTURE (16 BYTES)

0	WORD	STARTING LEFT-EDGE RELATIVE TO CONTAINER
2	WORD	STARTING TOP-EDGE RELATIVE TO CONTAINER
4	BYTE	FRONT PEN COLOR
5	BYTE	BACK PEN COLOR - UNUSED
6	BYTE	DRAW MODE - JAM1 OR XOR
7	BYTE	NUMBER OF PAIRS OF COORDINATES
8	LONG	POINTER TO A TABLE OF COORDINATES
12	LONG	POINTER TO NEXT BORDER STRUCTURE

Listing 1

```
;LISTING 1
;Mandelbrot/Julia Set - scale factor 2^9
  include exec_macros.i
  include intmacros.i
  include gfxmacros.i
  include dosmacros.i
  include mathffpmacros.i
  include menu.i
mul2 macro    ;d0*d1 -- d2,d3
     moveq   #7,d5
     moveq   #0,d7
     moveq   #0,d2
     moveq   #12,d5
mr1\@ asl.l  #1,d7
     roxl.l  #1,d2
     asl.l   #1,d0
     bcc     mr2\@
     add.l   d1,d3
     addx.l  d6,d2
mr2\@ dbf    d5,mr1\@
     endm
smul macro
   move.l   \1,d0
   move.l   \2,d1
   move.w   d0,d2
   mulu     d1,d2
   swap     d0
   mulu     d1,d0
   swap     d0
   clr.w    d0
   add.l    d2,d0
   endm

depth equ 5
pixels equ 32
maxcount equ 31
```

```
Bpsoc macro        ;[illegible]
    move.l    [illegible],a1
    move.w    [illegible],d0
    cmp.w     #[illegible],d0       ;adjust down
    sub.w     #[illegible],d0
    ext.l     d0
    ext.l     d1
    move.l    gfxbase,a6
    jsr       [illegible](a6)
    endm
mscale  macro    ;move d0 to [illegible]
  move.l     #[illegible],d7  ;[illegible]
  moveq      #[illegible],d1
  muldp                          ;scale it
  fixdp                          ;whole number in d0
  move.l     d0,\1               ;save it
  endm
beep    macro
  movea.l  screen,a0
  [illegible]  displaybeep
  endm

  move.l     sp,stack           ;save stack pointer
open_libs                        ;open all the libraries we
need
  openlib    int,done
  openlib    dos,close_int
  openlib    gfx,close_dos
  openlib    dmath,close_gfx

setup                            ;open a screen of 320 x 200
make_screen
  openscreen myscreen,close_libs
  moveq.w    #64,maxcount
make_window
  openwindow mywindow,close_screen
  palette    colormap,[illegible]
  mode       [illegible]
  openmenu   menu
msg_check
  [illegible] msg_check
  cmpi.l   #menupick,d0
  bne.s    msg_check
  eval_menunumber
  tst.w    d0
  bne.s    msg_check
  tst.w    d1
  beq.s    is_mandelbrot
  cmpi.w   #1,d1
  beq.s    is_julia
  bra.s    msg_check
is_mandelbrot
  move.w   #0,julia
  bra.s    coordinates
is_julia
  move.w   #1,julia              ;flag julia
coordinates
  removegadgets                  ;don't show gadgets
  lea      gadget1buffer,a0      ;X left
  convertdp                      ;make dp number
  movedp   d0,d6                 ;save in d6/d7
  mscale   xc
  lea      gadget2buffer,a0      ;X right
  convertdp
  movedp   d6,d2                 ;d2-d3 = X left
  subdp                          ;d0-d1 = X right - X left
  movedp   d0,d6                 ;save difference in d6/d7
  fltdp    320                   ;320 as dp number
  movedp   d0,d2                 ;move to d2/d3
  movedp   d6,d0                 ;move difference to d0/d1
  divdp                          ;difference / 320
  mscale   xscale

  lea      gadget4buffer,a0      ;Y bottom
  convertdp
  movedp   d0,d6
  mscale   yc
  lea      gadget3buffer,a0
  convertdp
  movedp   d6,d2
  subdp
  movedp   d0,d6
  fltdp    200
  movedp   d0,d2
  movedp   d6,d0
  divdp
  mscale   yscale

  lea      gadget5buffer,a0
  convertdp
  mscale   julian
  lea      gadget6buffer,a0
  convertdp
  mscale   julian
scaled_mandelbrot/julia_demo
check
  moveq.l  #0,a1
start
  [illegible]
  move.l   xc,xloc
  move.w   #0,across
mil
  move.l   yc,yloc
  move.w   #0,down
mil
  move.l   xloc,aloc
  move.l   yloc,bloc

  moveq.w  #0,count
again
  move.w   #0,sign
tempi
  move.l   aloc,d0               ;sign check
  bpl.s    squarea
  neg.l    d0
  move.l   d0,aloc
  addq.w   #1,sign
squarea
  move.l   d0,d1
  muls
  move.l   d2,asqr               ;save it
  move.l   d1,asqr
test
  move.l   bloc,d0
  bpl.s    squareb               ;sign check
  neg.l    d0
  move.l   d0,bloc
  addq.w   #1,sign
squareb
  move.l   d0,d1
  muls
  move.l   d0,temp               ;save it
  move.l   d1,temp
loop
```

```
   move.l   asqr,d0
   move.l   asqr4,d1
   add.l    d1,d3
   addx.l   d0,d2                  ;d2,d3 = asquare +
bsquare
   cmp.l    #$10000000,d2          ;compare to 4
   blo.s    imag                   ;branch if lower

   move.w   count,d0               ;count -> color
   andi.l   #31,d0                 ;optional color1
   ;andi.l  #15,d0                 ;optional color2
   ;addi.w  #16,d0
   ;lsr.l   #2,d0                  ;optional color3
   foreground                      ;set apen
   @pset    across,down            ;pset the point
   bra      fin
imag
   move.l   aloc,d0
   move.l   bloc,d1
   mult                            ;d2,d3 = aa * bb
   moveq    #28,d5                 ;scale / 2
   cmpi.w   #1,sign                ;check if negative
   bne.s    scale2                 ;branch if not
   neg.l    d3                     ; or change the value
   negx.l   d2                     ; of both registers
scale2
   asr.l    #1,d2                  ;shift
   roxr.l   #1,d3                  ; with carry
   subq.l   #1,d5
   bne.s    scale2
   tst.w    julia                  ;computing Julia Set?
   bne.s    add_juliab
   add.l    yloc,d3                ;guess not
   bra.s    scale2a
add_juliab
   add.l    juliab,d3              ;guess so
scale2a
   move.l   d3,bloc                ;new imaginary part

real
   move.l   asqr,d0
   move.l   asqr4,d1
   move.l   bsqr,d2
   move.l   bsqr4,d3
   sub.l    d3,d1
   subx.l   d2,d0                  ;d0 = asquare - bsquare
;realpart_ok!
   move.l   #28,d5
scale1
   asr.l    #1,d0
   roxr.l   #1,d1
   subq.l   #1,d5
   bne.s    scale1
   tst.w    julia
   bne      add_juliaa
   add.l    xloc,d1
   bra.s    scale1a
add_juliaa
   add.l    juliaa,d1              ; plus juliaa
scale1a
   move.l   d1,aloc                ;new real part

   addq.w   #1,count               ;increase count
   cmpa.w   maxcount,count         ;up to maximum yet ?
   bne      again                  ;branch if not
;  moveq.w  #17,d0                 ;optional Set coloring
;  foreground                      ; or leave it black
;  @pset    across,down            ; by skipping it
fin
   move.l   yloc,d0
   add.l    yscale,d0              ;increase yloc by yscale
   move.l   d0,yloc

check_for_message
   cfm      no_message
   cmpi.l   #menupick,d2           ;a menu?
   beq.s    check_menus
   cmpi.l   #mousebuttons,d2       ;press in?
   beq      zoom
   bra      no_message             ;nothing
check_menus
   eval_menunumber                 ;get menu,item,subitem #
   tst.w    d0                     ;menu0?
   beq.s    handle_menu0
   cmpi.w   #1,d0                  ;menu1?
   beq.s    handle_menu1
   bra      no_message
handle_menu0
   cmpi.w   #0,d1                  ;item0?
   beq.s    do_mandelbrot
   cmpi.w   #1,d1                  ;item1?
   beq.s    do_julia
   cmpi.w   #2,d1                  ;item2?
   beq.s    do_coordinates
   cmpi.w   #3,d1                  ;item3?
   beq      close_window
   bra      no_message
do_mandelbrot
   move.w   #0,julia               ;unflag julia
   bra      start
do_julia
   move.w   #1,julia               ;flag julia
   bra      start
do_coordinates
   pcls                            ;clear
   closemenu                       ; and
   closewindow                     ;  close
   bra      make_window            ;redisplay gadgets
handle_menu1
   tst.w    d1                     ;subitem0?
   beq.s    do_itcount
   bra      no_message
do_itcount
   tst.w    d2
   beq.s    do_set64
   cmpi.w   #1,d2
   beq.s    do_set128
   cmpi.w   #2,d2
   beq.s    do_set256
   cmpi.w   #3,d2
   beq.s    do_set512
   cmpi.w   #4,d2
   beq.s    do_set1024
   bra      no_message
do_set64
   movea.w  #64,maxcount           ;reset maxcount
   bra      no_message
do_set128
   movea.w  #128,maxcount
   bra      no_message
do_set256
```

(continued on page 66)

Porting a B+Tree Library to the Amiga...

by John Bushakra

Computer programmers are often faced with the daunting task of re-inventing the wheel. The users of our programs demand—rightfully—that they be able to share their information among all the different systems they may be using, and as programmers we must accommodate them in order to survive. But among programmers themselves, sharing information is quite a different story. As my father once told me, "The only people in this world who want you to have accurate information are the people you work for." Nowhere is this more true than in the computer software industry. To his observation, I can only add this parenthetical remark: sometimes, even the people you work for don't want you to have accurate information. In any event, the end result is that much of a computer programmer's time is spent re-inventing technology that someone else has already developed.

Anyway, this article isn't going to be a lecture about the evils of hoarding information. It's about porting a library of data management routines from the PC world to the Amiga world, and is meant to help other Amiga programmers, who might be sitting huddled in their caves, chiseling away on a programming tool that has already been implemented on the PC.

The product I'm referring to is called the C/Database Toolchest, and is available from Mix Software (the address is given at the end of this article). Mix Software is well known in the IBM world as the maker of a full blown C development environment which it sells for around—better sit down for this—$65. That price includes the compiler, source level debugger, library source code, a BCD business math package, and shipping. The C/Database Toolchest sells for $19.95, plus $10 for the C source code.

The product consists of a B+Tree library for managing index files, and an ISAM (Indexed Sequential Access Method) library, which manages both the indexing and data files needed by your application. There are numerous other utilities provided with the software. These include helpful debugging routines, functions for converting your data and index files to and from dBASE format, and compression routines—which essentially just physically remove deleted records from your data files. There is also a small database application called LDM, for Little Data Manager, which illustrates the use of the B+Tree and ISAM libraries. All source code is included with these utilities, including the LDM program. LDM uses typical character-based windowing functions and *Lotus*-style menu strips, but these could easily be replaced with their Amiga equivalents, to create a powerful database manager. A 350-page manual is also included with the package.

Presumably, a computer program is written with the intent of processing some type of information. The information of interest is often composed of several related data items, which are grouped together into data structures. When you design a data structure for your program, there will be one or two fields which you will want to use to identify a particular instance of the structure. The fields used for this purpose are called keys. For example, consider the following data structure:

```
struct person {
    char last_name[16], first_name[16];
    char address[32], phone[14], zip[10];
};
```

The last_name field is commonly used as a key field. Using this key, we might ask our database management software for a list of all the Smiths in our files. Or, if we used the zip field as a key, we could ask for all those people living within a certain zip-code.

Keys for a particular database are stored in an index file. An index, to quote the Toolchest manual, is "an ordered list containing pointers to data." The best illustration for this definition is the index in any text book. The index in a book is an ordered list of selected topics, and the page numbers given with each topic "point" you to the correct location in the book.

Every time a record is inserted into a data file, the data contained in the key fields of that record are written into the index file. Associated with each entry in the index file is a pointer, which shows where to look in the data file for the corresponding data record.

The records in the data file are stored in no particular order. The keys however, do have an order imposed on them. Ascending order using the ASCII collating scheme is frequently used, but most file management libraries, including the C/Database Toolchest, allow you to define your own key comparison functions. Keys provide us with a way to get different "views" of the information in our data files (for instance, all the Smiths living within a certain zip-code). But storing keys and data in separate files does not really give any increase in performance. We would still have to perform a sequential search of the index file to find the data records we're interested in.

The trick, then, is to devise an efficient way of storing keys in the index file. Here is where the B+Tree library in the C/Database Toolchest (CBT for short) comes in. The CBT library contains routines for managing the index files used by your application. The keys in the index are stored using an implementation of the B+Tree algorithm, so the index is organized in an efficient manner. Very briefly, a B-Tree is a generalization of another type of tree data structure called a 2-3 tree—so called because all of the interior nodes of the tree have either two or three children. A B+tree is yet another variation of the B-Tree, which, among other things, allows you to store variable length records by default.

Records in the CBT index file consist of keys and items. An item is defined to be a long integer and is typically used as an offset pointer into a second, sequentially organized file, which contains the actual information used by your application, for example, the names, addresses, phone numbers, and zip codes for each person data structure, defined above.

The CBT library contains many functions that can be used to create and maintain your index files. Among the different types of functions available are stepping forward or backward through the index, moving to the head or tail of the index, locating a particular key and item, and deleting a particular key and item. As mentioned earlier, the CBT library allows variable length keys by default. Other B-Tree packages I've seen require considerable additional work to use variable length records. One other nice feature of the library is that it allows multiple keys to be stored in a single index file. This means that if you wanted to use two different fields of the person data structure as keys, say, last_name and zip, you could store both of them in the same index file.

Since only long integers can be stored with the keys in the index file, you will probably never use the CBT library directly. Instead, the file management portion of your application will use the ISAM library, which is built on top of the CBT library. With the ISAM library, your data records can take any format needed by your application.

As noted above, ISAM stands for Indexed Sequential Access Method. This external storage algorithm gives us a way to organize data so that it can be manipulated in two ways—randomly, and sequentially. That is, we can access any record, regardless of its position in the database, or we can access the records in the order they are physically stored in the data file.

The ISAM library manages both the index file and the data file, so it contains routines for manipulating keys, which in turn call the CBT functions, as well as its own I/O functions for managing insertions, deletions, and searching for records in the data file. Among many others, there are functions provided for creating the database, opening an existing database, defining key fields, and making indexes.

The ISAM library supports many advanced features, including variable length records—by default, like the CBT library. The library also allows you to create indexes "on the fly. This feature allows the user of your program to define a temporary key field with which to retrieve records. For example, you the programmer might create indexes for the last_name and zip fields of the person data structure. One time, however, the user needs a report sorted on the city field. He would indicate this to your program, and by calling a single ISAM function, you can construct an index for him using the city field. All the records in the data file are immediately accessible by the new index. When the report is finished, you can remove the index, again using only a single function call.

The ISAM library also supports segmented keys, or keys that are made of several different fields from your data structure. Again, using our venerable person data structure, suppose the user wanted to generate a report which lists all the records sorted in ascending order by the state field, and within each state, the user also wants the zip code of each person sorted in ascending order. We can do this for him by concatenating the state and zip fields into one key, and then using these six files include the standard header files string.h and stdlib.h.

...Using Mix Software's C/Database Toolchest...

with double quotes instead of the usual angle brackets (< >). If the quotes aren't replaced with brackets, the C compiler will look in the current directory for these two header files, instead of the directory assigned to the symbol INCLUDE. The six offending files are bufpool.c, cbkey.c, ctrlrec.c, here.c, movekey.c, and btree.c.

The CBT library defines a data structure which it typedefs with the word Node. If you were never going to use the Exec Node structure, this wouldn't really cause any problems. It's highly, probable however, that you will be constructing lists of keys, and displaying them in those nifty scrolling list boxes that are so easy to make with the Amiga's new 2.0 operating system. The new list boxes make extensive use of Exec's List data structure, which of course uses Nodes to link the list together. In order to avoid conflicts, it's easier and safer to just give the CBT library Node structure a new name.

CBT's Node structure is referenced many times throughout the library, but we can easily change all the references at once, using the SPLAT utility that comes with the SAS C compiler. SPLAT is a program that takes a search pattern, a replacement string, and a set of files as input. When the search pattern is located in a file, it is changed to the replacement string. You will use SPLAT to search for all occurrences of the string "Node", and replace it with the string "CBNode". Only one command is needed to change all the CBT source files:

SPLAT -o Node CBNode #?.c #?.h #?.i

The -o option directs SPLAT to overwrite the original file. If you don't feel comfortable with this, then you can use the -d option to direct the output file to a different directory. I don't recommend letting SPLAT create files with the default .$$$ extension? AmigaDOS doesn't seem to like dollar signs as well as human beings.

Microsoft C uses a function called memmove() to copy bytes of memory from one location to another. There are two equivalents for this function in SAS C. The first is memcpy(), and the second is movmem(). When I compiled the CBT library with memcpy(), it simply refused to run. Unfortunately, I did not have a chance to research this problem fully. The solution is simply to use the movemem() function instead. You will have to change the order of the arguments, but that's not really a big deal. The source files that require this change are cbkey.c, here.c, and movekey.c.

Another Microsoft C memory function, memicmp(), is used in the CBT key comparison routine. The SAS C equivalent for memicmp() is memcmp(). The source file requiring this change is cbkeycmp.c.

The last change required before building the library is changing the Microsoft C make file to something that SAS C's LMK can digest. You will find a replacement file for LMK in listing one. Make files are used to describe the interdependencies of files that comprise a software application. Given a target, LMK examines the date and time stamp on the files that make up the target. Whenever a component of the target is found to be out-of-date, it is recompiled.

In this case, the target of the build is the library, CBT.LIB, and the components that make up the target are the object files produced by the compiler. Whenever the date and time stamp on the object file is older than the date and time stamp on the corresponding source file (i.e., the source file was modified more recently than the object file), the compiler is invoked to rebuild the object file. To make things more efficient, you can use the -R option, which instructs the compiler to replace the out-of-date object file in the specified library with the new one. Since LMK is smart enough to compile only the source files that are outdated, the -R option will allow us to only update the library for those object files that have been recompiled. Otherwise, we'd have to keep all the object files around on the hard disk, and rebuild the library from scratch, even if only one source file was modified.

To build the CBT library, type

LMK -f cbt.mak

When the process is done, you will find the CBT.LIB file in your LIB: directory.

Microsoft C uses a function called memmove() to copy bytes of memory from one location to another. There are two equivalents for this function in SAS C. The first is memcpy(), and the second is movmem().

The changes that need to be made to the ISAM library are equally simple. Like the CBT library, the easiest place to start porting the ISAM library is with changing some #include statements.

Some of the ISAM source files include files from the CBT library. Therefore, you will have to change the #include statements in these files, so the compiler will look in the directory into which you copied the CBT library source files. The ISAM files that require this change are? isam.i, isam.h, and createdb.c.

The level one I/O routines in Microsoft C require an include file called io.h. This file does not exist under SAS C, and this include must be changed to the file fcntl.h. There are seven files requiring this change, and they are addrec.c, closedb.c, createdb.c, getrec.c, getrlen.c, holes.c, and opendb.c.

When opening a database with the ISAM library, the only parameter required is the file name, without an extension. Both the open and create database functions automatically provide an extension of .IDX, and .DB for the index file and data file, respectively. The library functions go to great pains to strip an existing extension off the file name passed to them before opening or creating the database. Because AmigaDOS is less fussy about file names than MS-DOS, stripping an existing extension off the file name simply isn't needed. We can make the open and create functions somewhat simpler by

reworking them so that they only concatenate an extension of .IDX for the index file, and .DB for the data file. Since AmigaDOS allows multiple periods to appear in filenames, we can ignore an existing extension with no worries.

Begin this change by bringing up a file called filename.c in your text editor. There are two very similar functions defined in this file—one makes the data file name, and the other makes the index file name. Change these two functions so that all they do is strcpy() the filename passed in by the caller into the filename_buf buffer, then call the function chg_extent(), with the buffer, and the desired extension (either .DB or .IDX).

Also note that the default extensions .DB and .IDX are #defined in this file—these extensions can be changed to suit your needs.

Notice at the top of filename.c a file called filename.h is included. This header file defines the maximum number of characters allowed in a drive prefix, path name, file name, and extension. They are all specified in terms of MS-DOS limitations, and need to be changed to the AmigaDOS equivalents. Specifically, a disk drive prefix is four characters (e.g., DH0:), a path name can be up to 256 characters, and a filename can be up to 31 characters. The default extension size is defined to be four characters (three letters, plus the dot). I elected not to change this, since I didn't change the default .DB and .IDX extensions. If you change these extensions, be sure to increase the size of the maximum file name extension accordingly.

If you look at the two functions in filename.c again, you will notice that they both call a function that appends the default extension to the file name. This function is defined in a file called chgextnr.c. Bring this file up in your text editor, and comment out the for() loop in the function chg_extent(). Rework the routine so that all it does is strcat() the extension onto the end of the file name buffer, and then return.

In making these changes, we've effectively bypassed the function defined in the file path.c. Therefore, we can remove this file from the make file provided for the ISAM library.

The next change is going to be a little difficult to explain, since I can't reproduce any of the ISAM source code for you in a listing. On a PC, when you declare an integer variable with only the int keyword, the compiler gives you 16 bits in which to store your integer. On the Amiga, the same declaration generates a 32-bit variable. Because of this difference, some changes are required to the routines that manage deleted records in the data file. These routines are defined in the file holes.c.

Bring this file up in your editor, and look at the function l_mk_size_key(). It's easy to see that this function operates on a 16-bit integer. First the upper eight bits are shifted right, and "masked" with 0xff, then the lower eight bits are masked with 0xff. This function needs to be changed so that it operates on a thirty-two bit integer.

First, change the function to shift the size parameter to the right by 24 bits and mask it with 0xff. Then shift it by 16 bits, and apply the mask. Shift it again by eight bits and apply the mask, and finally, apply the mask to the unshifted size parameter to get the lower eight bits. In short, make the function l_mk_size_key() look just like the function l_mk_offset_key(), with the obvious exception that l_mk_size_key() operates on the size parameter.

A similar change needs to be made to the function l_get_size(), in the same file. Like l_mk_size_key(), this function operates on a 16-bit integer, rather than 32-bit integer. To make a long story short, make l_get_size() look exactly like l_get_offset(), except that l_get_size() sets the size variable, not the offset variable.

The level one I/O routine open() accepts a file protection parameter, although the SAS C manual states that this parameter is ignored. Still, if you don't provide it, you will get warning messages when you compile the library. There are occurrences of the open() function in both opendb.c, and createdb.c. You can simply specify a zero as the last parameter to open(), and the compiler will be happy and content.

Another Microsoft C memory function, memicmp(), is used in the CBT key comparison routine. The SAS C equivalent for memicmp() is memcmp(). The source file requiring this change is cbkeycmp.c.

As its name implies, the file createdb.c has routines that create and initialize the database index and data files. Bring this file up in your editor and look at the function l_init_header(). The first thing this function does is lseek() eight bytes past the beginning of the freshly created data file. That might be just dandy in MS-DOS, but AmigaDOS will not be pleased at all if you try to seek past the beginning of an empty file. The call to lseek() needs to be commented out, and replaced with two write() statements that write out dummy values for the two variables, strings_length, and field_count. After the write() statements, the file marker will be positioned at the same place it would have been, had the lseek() call succeeded. When you insert the two write() statements, be sure to follow Mix Software's convention of checking the return value (in this case, the number of bytes written). If the value is not four (the size of a long integer), something is wrong, and you need to return the error value E_IO immediately.

After searching all the source files for the obvious changes given above, I compiled the ISAM source files, and built the library. My initial tests, creating, opening, inserting records and retrieving records were successful, but then something disturbing happened. After deleting a record, closing the database, and then re-opening the

database, the indexes I had defined seemed to have disappeared. A dump of the index file proved they were in fact still there, but the iopen_db() function would not read them. Managing deleted records is a major, complicated part of the ISAM library—no doubt this problem was not going to be easy to track down.

Whenever you insert a record in the data file, the data in the key field(s) of that record are also written in the index file. But keys aren't the only things stored in the index file. When you define an index, the name of that index is also written in the index file. Furthermore, whenever you delete a record, a key and item pair is written to the index file that tells where the "hole" is in the data file so that space can be re-used later. How does the ISAM library tell the difference between all of these key and item pairs in the index file? By prefixing each key with a signed byte that identifies the key as either the name of an index, a pointer to a hole in the data file, or an actual key that points to a valid record.

When you open up a database, the I_next_index() routine (in opendb.c) starts reading key/item pairs from the index file, trying to find all the names of the indexes you have defined. When it sees a key with a prefix byte that identifies that key as something other than the name of an index, it stops, thinking it has examined all the keys in the index file.

In this case, I_next_index() couldn't have been more wrong. The first key/item pair it was reading from the index file happened to be a pointer to a hole in the data file. It saw the prefix byte identifying it as such, and quit, before it even saw any index names.

Obviously, I can't provide you with the exact source code for I_next_index(). Fortunately, the changes that need to be made aren't that difficult. The function should remain basically the same—all parameters passed to it, and its local variables do not need to be changed. What the function should do is read keys one at a time, until it finds one with a zero byte prepended to it (the zero byte identifies the key as the name of an index). When such a key is found, it should return to its caller (I_iopen()). The caller will do whatever processing is necessary on the returned key, and continue to call I_next_index() until it returns the value of EOI (for "end of index"). The down side of this solution is when the index file becomes very large, it could take some time to open the database. On the positive side, opening the database is something you are likely to do only once; afterward, you still gain all the speed benefits of the B+Tree storage algorithm. I'm not sure why this anomaly exists in the AmigaDOS port of the ISAM library. I've since run the LDM program on a PC, and experienced no such problems when deleting records.

Listing 2 has the make file for the ISAM library. There are two source files on the distribution diskette that use ROM BIOS interrupts to get the disk drive number and current directory. These two files aren't needed at all for AmigaDOS, and can be removed from the make file. To build the library, type:

LMK -f isam.mak

When the build is complete, you will find the file ISAM.LIB in the directory assigned to LIB:.

Overall, I'm very impressed with the quality of the C/Database Toolchest. The entire system is well designed and coded. The 350-page manual that accompanies the package is a gem; it easily ranks among the best software documentation I've ever seen, on any platform. In addition to thoroughly explaining all of the functions in the libraries, it also contains a chapter with general information on the Indexed Sequential Access Method, and still another chapter on external tree data structures. This manual is well structured, well written, and informative as well as educational? indeed a rare find in today's software market.

The C/Database Toolchest seems to be right at home on the Amiga. It's a powerful and badly needed addition to any programmer's library. The moral of the story is, that if you can't find a programming tool you need to complete a project on the Amiga, come out of your cave and investigate the possibility of porting existing code. And if you do find something that is portable, by all means share the information with your fellow coders!

Mix Software
1132 Commerce Drive
Richardson, TX 75081
1-800-333-0330

Listing 1

```
# Listing One.  LMK file for building the CVT library.
# John Buchanan  06/10/91
#
#


CC=lc
MAKE=-$(LIB)dos.lib
LIB=LIB:chc.lib
OBJECTS=cvtstr.o turbo.o ctisc.o btree.o \
        db_rec.o chkrec.o chk_rec.cfg \
        buffpool.o member.o

$(LIB): $(LIB)isamdb.o cvtstr.o cvtdtc.o cbclose.o \
        cbflush.o cbfindkf.o cbflush.o cbindmrk.o cbmod.o \
        cbnew.o cbseek.o cbsysinq.o cbmark.o cbmodify.o \
        nodelete.o cbmet.o cbopen.o cbprev.o cbcrtkey.o \
        cbcommq.o cbretmsg.o cbinsmsg.o cbtell.o \
cbutime.o \
        isutnode.o cbcmpkey.o cbdelkey.o cbgpsel.o \
cbclass.o \
        cbinsert.o tree.o keyvalid.o makefil.o offleft.o \
        offright.o insnode.o delnode.o isutnode.o \
cbutbuic.o \
        ctisc.o movekey.o cbent.o getlock.o btree.o \
mem.o \
        cvtsyscm.o node.o block.o buffpool.o member.o

cbinit.o: cbinit.c {HEADERS}
    $(CC) $(CFLAGS) cbinit.c
```

```
cbcreate.o: cbcreate.c $(HEADERS)
    $(CC) $(FLAGS) cbcreate.c

cbcurr.o: cbcurr.c $(HEADERS)
    $(CC) $(FLAGS) cbcurr.c

cbexit.o: cbexit.c $(HEADERS)
    $(CC) $(FLAGS) cbexit.c

cbclose.o: cbclose.c $(HEADERS)
    $(CC) $(FLAGS) cbclose.c

cbfind.o: cbfind.c $(HEADERS)
    $(CC) $(FLAGS) cbfind.c

cbfindit.o: cbfindit.c $(HEADERS)
    $(CC) $(FLAGS) cbfindit.c

cbflush.o: cbflush.c $(HEADERS)
    $(CC) $(FLAGS) cbflush.c

cbfndmrk.o: cbfndmrk.c $(HEADERS)
    $(CC) $(FLAGS) cbfndmrk.c

cbhead.o: cbhead.c $(HEADERS)
    $(CC) $(FLAGS) cbhead.c

cbkey.o: cbkey.c $(HEADERS)
    $(CC) $(FLAGS) cbkey.c

cbdepth.o: cbdepth.c $(HEADERS)
    $(CC) $(FLAGS) cbdepth.c

cbkeylen.o: cbkeylen.c $(HEADERS)
    $(CC) $(FLAGS) cbkeylen.c

cbmark.o: cbmark.c $(HEADERS)
    $(CC) $(FLAGS) cbmark.c

cbmodify.o: cbmodify.c $(HEADERS)
    $(CC) $(FLAGS) cbmodify.c

cbdelete.o: cbdelete.c $(HEADERS)
    $(CC) $(FLAGS) cbdelete.c

cbnext.o: cbnext.c $(HEADERS)
    $(CC) $(FLAGS) cbnext.c

cbopen.o: cbopen.c $(HEADERS)
    $(CC) $(FLAGS) cbopen.c

cbprev.o: cbprev.c $(HEADERS)
    $(CC) $(FLAGS) cbprev.c

cbpriter.o: cbpriter.c $(HEADERS)
    $(CC) $(FLAGS) cbpriter.c

cbcommsg.o: cbcommsg.c $(HEADERS)
    $(CC) $(FLAGS) cbcommsg.c

cberrmsg.o: cberrmsg.c $(HEADERS)
    $(CC) $(FLAGS) cberrmsg.c

cbfermsg.o: cbfermsg.c $(HEADERS)
    $(CC) $(FLAGS) cbfermsg.c

cbtail.o: cbtail.c $(HEADERS)
```



```
    $(CC) $(FLAGS) cbtail.c

curitem.o: curitem.c $(HEADERS)
    $(CC) $(FLAGS) curitem.c

isdecml.o: isdecml.c $(HEADERS)
    $(CC) $(FLAGS) isdecml.c

posnext.o: posnext.c $(HEADERS)
    $(CC) $(FLAGS) posnext.c

posfirst.o: posfirst.c $(HEADERS)
    $(CC) $(FLAGS) posfirst.c

pasprev.o: pasprev.c $(HEADERS)
    $(CC) $(FLAGS) pasprev.c

posinset.o: posinset.c $(HEADERS)
    $(CC) $(FLAGS) posinset.c

cbinsert.o: cbinsert.c $(HEADERS)
    $(CC) $(FLAGS) cbinsert.c

root.o: root.c $(HEADERS)
    $(CC) $(FLAGS) root.c

keyvalid.o: keyvalid.c $(HEADERS)
    $(CC) $(FLAGS) keyvalid.c

makelit.o: makelit.c $(HEADERS)
    $(CC) $(FLAGS) makelit.c
```

```
offleft.o: offleft.c $(HEADERS)
	$(CC) $(FLAGS) offleft.c

offright.o: offright.c $(HEADERS)
	$(CC) $(FLAGS) offright.c

fmknode.o: fmknode.c $(HEADERS)
	$(CC) $(FLAGS) fmknode.c

delnode.o: delnode.c $(HEADERS)
	$(CC) $(FLAGS) delnode.c

joinnode.o: joinnode.c $(HEADERS)
	$(CC) $(FLAGS) joinnode.c

cbdikeit.o: cbdikeit.c $(HEADERS)
	$(CC) $(FLAGS) cbdikeit.c

crlrec.o: crlrec.c $(HEADERS)
	$(CC) $(FLAGS) crlrec.c

movekey.o: movekey.c $(HEADERS)
	$(CC) $(FLAGS) movekey.c

space.o: space.c $(HEADERS)
	$(CC) $(FLAGS) space.c

getblock.o: getblock.c $(HEADERS)
	$(CC) $(FLAGS) getblock.c

btree.o: btree.c $(HEADERS)
	$(CC) $(FLAGS) btree.c

here.o: here.c $(HEADERS)
	$(CC) $(FLAGS) here.c

cbkeycmp.o: cbkeycmp.c $(HEADERS)
	$(CC) $(FLAGS) cbkeycmp.c

node.o: node.c $(HEADERS)
	$(CC) $(FLAGS) node.c

blockio.o: blockio.c $(HEADERS)
	$(CC) $(FLAGS) blockio.c

putpool.o: putpool.c $(HEADERS)
	$(CC) $(FLAGS) putpool.c

member.o: member.c $(HEADERS)
	$(CC) $(FLAGS) member.c
```

Listing 2

```
* Listing Two.  LMK file for building
the ISAM library
# John Bushakra   06/10/91
#


# Removed path.c, getdisk.c, and getcurdr.c
for AMIGA conversion
#
CC=lc
FLAGS=-Rlib:isam.lib
LIB=LIB:isam.lib
HEADERS= isam.i isam.h isamelt.h isam.cfg \
        dh0:lc/source/cbt/cbtree.h dh0:lc/
source/cbt/member.h

$(LIB):  isaminit.o isamexit.o copydb.o
newindex.o destrydb.o \
        renamedb.o dbhandle.o findrec.o
findtail.o getfldct.o \
        getidxnm.o isammsg.o prterr.o
matchkey.o modrec.o \
        rmindex.o delrec.o holes.o
showkey.o addrec.o \
        createdb.o findkey.o findprev.o
mkindex.o namelist.o \
        progress.o ihandle.o showdb.o
findmark.o findnext.o \
        findhead.o getrec.o getrlen.o
markrec.o matpre.o \
        showdesc.o getdesc.o showtid.o
getfldnm.o showidx.o \
        showrec.o upindex.o mkkey.o
opendb.o closedb.o \
        filename.o chgextnt.o flushdb.o

isaminit.o: isaminit.c $(HEADERS)
    $(CC) $(FLAGS)  isaminit.c

isamexit.o: isamexit.c $(HEADERS)
    $(CC) $(FLAGS)    isamexit.c

copydb.o: copydb.c $(HEADERS)
    $(CC) $(FLAGS)    copydb.c

newindex.o: newindex.c $(HEADERS)
    $(CC) $(FLAGS)    newindex.c
```

```
destrydb.o: destrydb.c $(HEADERS)
	$(CC) $(FLAGS)	destrydb.c

renamedb.o: renamedb.c $(HEADERS)
	$(CC) $(FLAGS)	renamedb.c

dbhandle.o: dbhandle.c $(HEADERS)
	$(CC) $(FLAGS)	dbhandle.c

findrec.o: findrec.c $(HEADERS)
	$(CC) $(FLAGS)	findrec.c

findtail.o: findtail.c $(HEADERS)
	$(CC) $(FLAGS)	findtail.c

getfldct.o: getfldct.c $(HEADERS)
	$(CC) $(FLAGS)	getfldct.c

getidxnm.o: getidxnm.c $(HEADERS)
	$(CC) $(FLAGS)	getidxnm.c

isammsg.o: isammsg.c $(HEADERS)
	$(CC) $(FLAGS)	isammsg.c

prterr.o: prterr.c $(HEADERS)
	$(CC) $(FLAGS)	prterr.c

matchkey.o: matchkey.c $(HEADERS)
	$(CC) $(FLAGS)	matchkey.c

modrec.o: modrec.c $(HEADERS)
	$(CC) $(FLAGS)	modrec.c

rmindex.o: rmindex.c $(HEADERS)
	$(CC) $(FLAGS)	rmindex.c

delrec.o: delrec.c $(HEADERS)
	$(CC) $(FLAGS)	delrec.c

holes.o: holes.c $(HEADERS)
	$(CC) $(FLAGS)	holes.c

showkey.o: showkey.c $(HEADERS)
	$(CC) $(FLAGS)	showkey.c

addrec.o: addrec.c $(HEADERS)
	$(CC) $(FLAGS)	addrec.c

createdb.o: createdb.c $(HEADERS)
	$(CC) $(FLAGS)	createdb.c

findkey.o: findkey.c $(HEADERS)
	$(CC) $(FLAGS)	findkey.c

findprev.o: findprev.c $(HEADERS)
	$(CC) $(FLAGS)	findprev.c

mkindex.o: mkindex.c $(HEADERS)
	$(CC) $(FLAGS)	mkindex.c

namelist.o: namelist.c $(HEADERS)
	$(CC) $(FLAGS)	namelist.c

progress.o: progress.c $(HEADERS)
	$(CC) $(FLAGS)	progress.c

ihandle.o: ihandle.c $(HEADERS)
	$(CC) $(FLAGS)	ihandle.c

showdb.o: showdb.c $(HEADERS)
	$(CC) $(FLAGS)	showdb.c

findmark.o: findmark.c $(HEADERS)
	$(CC) $(FLAGS)	findmark.c

findnext.o: findnext.c $(HEADERS)
	$(CC) $(FLAGS)	findnext.c

findhead.o: findhead.c $(HEADERS)
	$(CC) $(FLAGS)	findhead.c

getrec.o: getrec.c $(HEADERS)
	$(CC) $(FLAGS)	getrec.c

getrlen.o: getrlen.c $(HEADERS)
	$(CC) $(FLAGS)	getrlen.c

markrec.o: markrec.c $(HEADERS)
	$(CC) $(FLAGS)	markrec.c

matpre.o: matpre.c $(HEADERS)
	$(CC) $(FLAGS)	matpre.c

showdesc.o: showdesc.c $(HEADERS)
	$(CC) $(FLAGS)	showdesc.c

getdesc.o: getdesc.c $(HEADERS)
	$(CC) $(FLAGS)	getdesc.c

showfld.o: showfld.c $(HEADERS)
	$(CC) $(FLAGS)	showfld.c

getfldnm.o: getfldnm.c $(HEADERS)
	$(CC) $(FLAGS)	getfldnm.c

showidx.o: showidx.c $(HEADERS)
	$(CC) $(FLAGS)	showidx.c
```

(continued on page 71)

Who
can you turn to
for the
best coverage
of the fast-paced
Amiga
market?

Amazing Computing of course!

Amazing Computing for the Commodore Amiga, AC's GUIDE, and AC's TECH provide you with the most comprehensive coverage of the Amiga. Coverage you would expect from the longest running monthly Amiga publication.

The pages of Amazing Computing bring you insights into the world of the Commodore Amiga. You'll find comprehensive reviews of Amiga products, complete coverage of all the major Amiga trade shows, and hints, tips, and tutorials on a variety of Amiga subjects such as desktop publishing, video, programming, & hardware. You'll also find a listing of the latest Fred Fish disks, monthly columns on using the CLI and working with ARexx, and you can keep up to date with new releases in New Products and other neat stuff.

AC's GUIDE to the Commodore Amiga is an indispensable catalog of all the hardware, software, public domain collections, services and information available for the Amiga. This amazing book lists over 3500 products and is updated every six months!

AC's TECH for the Commodore Amiga provides the Amiga user with valuable insights into the inner workings of the Amiga. In-depth articles on programming and hardware enhancement are designed to help the user gain the knowledge he needs to get the most out of his machine.

Call 1-800-345-3360

Wrapped Up with True BASIC

by Roy M. Nuzzo

This article will, as a minimum, show you how to do a graphic word wrap and graphic-oriented text with full justification in code easily translated to any language. It uses True BASIC. This snippet of code is just an introduction. It is part of a much bigger and more serious application which can be further revealed if there is sufficient interest.

I am a heuristic style programmer who has passed from punch card Fortran in the early 60's to PCP11 front panel programming to mainframe multitasking Fortran - assembly and various Pascals on to 'C'. I was always of the 'First, Program in English' school. I have known many super programmers who were testing ideas for applications in Applesoft, or something like it, like me, behind closed doors. I have found that for idea development speed and power and porting, nothing, except English, beats True BASIC. Code that runs on my Tandy lap top or on a Mac, runs on the Amiga without any modification. This especially includes my heavily graphic oriented programs.

True BASIC code is inherently easy to translate. I set up my Manx 'C' source level debugger with a True BASIC window to test ideas on the fly, and use it as my 'C' text editor. It is a highly structured modular and compiled language. It is fast in execution. You may write subroutines in assembly or in 'C' and just use the True BASIC as a front end. You can also use any system call from within True BASIC that might be called in 'C' such as:

```
StringBuffer = AllocRemember(RememberPtr, BufferSize, MEMFAST)

with minimal change:

let StringBuffer = AllocRemember(RememberPtr, BufferSize, MEMFAST)
```

The NOLET option removes even this slight difference, but the let statement has many advantages in readability and locating assignments, so I avoid the 'NOLET' option.

There is no speed penalty for system calls done this way. You could learn 'C' like intuition programming from within True BASIC without ever having a 'C' compiler. In fact, I have found it easier to use system functions from True BASIC than from 'C'. There is no speed loss.

Platforms

The original Amiga offering of True BASIC works on Amiga 500, 1000, and 2000+ series under DOS 1.3 or 2.0+, all of which I have used. The originally marketed version does not work with 32-bit architecture as was the case with many products compiled for the original Amiga architecture specs. The 32-bit version, which I am currently using works on all Amigas (I have 6 different configurations covering all the above. It is being called a 'student version'. Why? Beats me. It is a complete and vastly expanded implementation of True BASIC. There is more in this 'student version' than in any language that I have seen. The only missing element is the 'Binder' better known as linker which allows you to tweak your written program to stand alone and not require the language system to run it. But that was always an independent offering.

The new version supports static array and subroutine memory, scripts, preloading compiled work code and many other nifty things to thoroughly spoil a programmer. Some of the code to be used in this series was written and tested on a Tandy lap top and tested under MSDOS even though it was written to be implemented on an Amiga.

Although the language is consistent across platforms, it does an excellent job at getting at details specific to the machine via the Toolkit (support of machine specific functions and libraries such as exec, diskfontlib, intuition etc.). That power is demonstrated in this particular project.

Game Plan

Here is the overall idea. Take nearly any kind of image and put it into a window, in any quadrant, with text, from any text file, word wrapping around it.

The pop-up window can be any legal size, the text can be, thanks to Paul Castonguay (who wrote a nice DiskFont library), shown in any font (including proportional) and be of any spacing and leading. The screen can be of any resolution and mode kind including EHB and, especially, HAM. Text should auto justify and auto hyphenate, when you want it to, and do so on the fly. Text should always be easily read regardless of the current color palette and particularly when running in HAM mode.

You should be able to defeat flicker, even when in interlace mode with 3-D glasses. Regardless of screen resolution, single or ANIM images must also allow true 3-D presentation (XSpecs) taken from live or computer generated sources. 3-D images ought to be showable on or alongside other non 3-D images and in normal program screens used for other program output. 3-D images ought to allow ANIMation and data query.

Text must scroll, with key press, forward and backward. The images, if they are ANIMs, must allow keyboard activation of animation, as well as stepwise single image advance or backup, independent of the text scrolling. User interaction with text or imagery must be handled. For example, a user ought to be able to step through

How to do a graphic word wrap and graphic-oriented, full text justification with True BASIC code

an ANIM (taken from video, say) and stop on any frame and point to anything in that frame to ask what it is or identify it in response to a text question. The image carries data that identifies that point and click for what it is.

The images, themselves, must be able to contain information which is pertinent to what they show (If you show a duck you might want to encode how much it weighs or how loud it quacks without an additional text file. A geographic map image ought to be able to carry the fact that Texas is color 4 and New Jersey is color 6 etc.).

User display programs ought to be able to query images for more than what is displayed. Labels to details seen in the images ought to be able to be shown and disappear independent of the user programming, and be accurately placed regardless of where, on screen, the image is shown. All frames of an ANIMation ought to be able carry hidden information about what is shown.

All of this ability ought to be available to a basic program as:

1. Load an image.
2. Ask the image what text file it wants, if any.
3. Load the text file (any size).
4. Ask the text file if there are there questions to be asked?
5. Present image and text in a pop-up window that does its own screen repair.
6. If the text asked questions, check the user answers against the correct ones which are stored invisibly within the text or within the image.
7. Tell the user the correct answers and score. Repair the screen.

Do all of this in under 15 program lines of code.

The last part first. You put all your programming power into compiled libraries, not programs. Each sub and definition ought to stand alone for its functionality. Doing so, simple single-line calls do complex things. A later improvement within a single library subroutine will be reflected in every program ever written by you that references that library. If I find a better way to hyphenate, every program which calls for hyphenation will be improved, including programs which I have forgotten about.

A good rule, if you are having a hard time deciding what to name a sub, chances are that it is a faulty sub. It should do one thing, making the name obvious. If you are tempted to place the word 'AND' in a subroutine name then split the sub.

We begin with a library to wrap text around brush images stored in array form. The source code to ABrushTextWrapLib is included on disk along with a detailed documentation file ABrushTextWrapLib.DOC. There are, in this library, 21 main subroutines and 5 definitions which do most of the work mentioned above (and other things).

Most? Well, ABrushTextWrapLib calls more essential functions from 3 other libs. Two are the familiar (to system programmers) 'amiga' and 'exec' libs and one is my own 'ScreenModeLib' (which handles extended screen mode switching and array-brush structure). These 3 libraries, called by my library, in turn call 4 more primitive libraries for deep and dirty primordial ooze functionality.

A few things about True BASIC:

Example: a simple function in this library, PolarCharWidth, returns the width of a character in current screen terms. Because you can designate the left, right, bottom, and top window boundaries to be anything in True BASIC (left edge can be minus pi and right edge can be Avogadro's number) you want to know the width of a printed character in terms of the current user declared screen dimensions. You declare the values of the screen edges and ask the system to tell you the number of characters that fit the entire screen independent of that. Divide to get a single character width. That width can be negative or positive depending on whether you set the window boundaries with the positive direction going rightward or leftward.

A very nice feature of this flexible window size value is that you can zoom images without changing the data. Why recalculate the values of 1000 points for x,y, & z when you can just change the values of the window edges? Four numbers, or easier yet:

```
Set Window Wleft * Zoom, Wright * Zoom, Wbot * Zoom, Wtop * Zoom
```

The space you allow for text and the amount of text you wish to show in that space, often conflict. You now apply some rules. If there is extra room to the right of the text, do we pad pixels to spread text and fill it? Well only if it doesn't stretch too few words over too much area. Apply a filter that pleases the eye. Filter? Any set of test rules which please by limiting an otherwise rigid rule. Such as "not if only two words" or "not if text takes up less than 75% of the space" etc. Anything you like.

A step into heuristic programming

If the text on a given line would extend too far and exceed the allotted space, break off a part that does fit. It is best is to break sentences at spaces. Hyphenation is trickier because it defies simple rules by way of numerous exceptions. I'm no linguist. I took the heuristic approach. Do tons of it by hand and TABULATE how breaks

actually split words. Group the results into as few categories as will cover the bulk of cases and make a lookup table to reflect this empiric result. Nice. You do not need to understand a phenomenon to deal with it. Just keep score. This is not cheating. This is life. We live and breathe by observation and correlation.

Here is my fast (just the most common splits) hyphenation scheme:

Separate spans of letters of four sizes by nontext control code separators shown here as '^':

1: "ING^^^OID^^^TIO^^^NES^^^MEN^^^LIN^^^TAC" SIZE 3
2: "SUB PRE NON
DIS^^^TRAN^^^OVER^^^NDER^^^UPER^^^POST" SIZE 4
3: "BKMNRWZHCFDTPJLVXY" with any two letters preceding SIZE 3 of 2
4: "GKMPQWRJCVDTSL" with any two letters following 2 of 2

Start looking from the right side of the line beginning at the point at which it exceeds the available space.

Oh by the way

An aside, about strings. True BASIC string handling is furiously fast. The form is familiar to 'C' programmers who use pointers.

Letters in a string are numbered from 1 to whatever, no limit. Zero precedes the first letter and MAXNUM means after the last (a built-in value equal to the largest number that the current machine can handle in hardware).

String$[J;K] means the text from character position number J to position number K.
"ABCDEFGHIJKL"[3:5] means "CDE"
"ABCDEFG"[0:MAXNUM] = "ABCDEFG" as does
"ABCDEFG"[0:1000], any number too large grabs up to the last character.

Given that S$ = "ABCDE", let S$[0:0] = "123" produces S$ = "123ABCDE"
Further, let S$[MAXNUM:0] = "789" now produces S$ = "123ABCDE789"
Yet further, let S$[4:7] = "" results in "123E789".

To check for a three letter sequence "ABC" in a string S$ from the right:

```
      for J = len(S$) to 3 step -1
   if pos( s$(J-2:J),"ABC" )   > 0 then
     call Sub("I found it.")
     exit for
   end if
next J
```

About numbers, do not worry about 'types' of numbers (INTEGER or FLOAT etc.). True BASIC figures them out from context. There are just strings and numbers. Period. Arrays can have any base.

You can have an array of five numbers starting from base -3 going to 1.

```
dim A(-3 to 1) also written as dim A(-3:1)
dim Vec(7) by default is from 1 to 7 unless you DAD
SET OPTION BASE 0 in which case it is from 0 to 7.
      for T = lbound(TextArray$) to ubound(TextArray$)
   if UCASE$( TextArray$(T)[1:1] ) = "ZEM" then
     call printtext(TextArray$(T), "blue")
   else
     call printtext(TextArray$(T), "black")
   end if
next T
```

Back to hyphenation:

Say the sentence is "Many have been hurt by inflammatory predesignations about their ...blah blah."

If predesignations only fits as far as predesignatio' then counting backward for three-letter sequences, from group one, find= "TIO" matches as a hyphenation break point. If we simply want to break as far to the right as possible, then this line would hyphenate 'predesigna-' and 'tion ' would start the next line. Some schemes do this rather than first look for a 'close enough' space break. To prevent too many hyphenations, some count how many and disallow hyphenations on successive lines, every third, etc. We will instead declare a larger zone for preferred line chopping and give space chopping first shot even if it is further left than a hyphenation break. Don't like that? Change it. If space chop fails then hyphenation tries. If that fails, the attempt is repeated for both, but further left.

A four letter breakpoint, from group two, lies further to the left "PRE". Even so, a space chop will be used if the initial zone is set so that it reaches the space before 'pre'. Failing that, break between any two letters composed of one from group 3 and the next from group 4.

Recap: Always break at a space if encountered. Test a larger distance from the right for spaces before trying to hyphenate. If there are no spaces up to this first limit point,look for hyphenation breakpoints instead. If that fails, look further yet to the left for spaces, and if failing look more yet for hyphenation. Last resort is to coldly truncate. In all cases one space is allowed for the hyphen.

Therefore, we need two subs to divide a line of text. One by space and one by hyphenation. Both confine themselves to the right sided range passed to them. Both are called by a control sub with increasing ranges of text from the right margin (if line division is not successful).

```
                ! sub SpaceChop(RetLine)
Control[Sub>
        ! sub DoHyphen(RetLine)
```

The calling control sub first takes care of any exceptional cases where chopping is not to occur and traps embedded code in the text that requests a line feed or form feed (user request, a yet higher priority).

Using an array of strings to hold various line or form feed sequences allows user designation of personal codes as well as those hard coded into the scheme. The hard coded ones are CHR$(10), CHR$(12), and CHR$(13). They are hard coded by simply priming the array with them. Next year you might consider adding others. Code modularity makes it easy to find where and remember how.

```
sub SetLineFeedArray(LFA$(),Vec(),Vec(#))
  ! This sub sets up and primes an array to hold
  ! line and form feeds
  ! To alter built in LF or FF everywhere, edit here.
  ! Remember the lower bound of the array is not
  ! limited to zero.
  !-----------------------
  set LFA$ = Null$(5)   ! 1 & 5 = USER, null is ignored
            let LFA$(5) = "LFFFUSER" ! UNUSED flag to tell sub that
                        ! array is primed.
            ! LINE FEED CHOICE
  let LFA$(1) = Text1FF$  ! USER CHOICE
  let LFA$(2) = CHR$(10)
  let LFA$(3) = CHR$(13)
            ! FORM FEED CHOICE
  let LFA$(4) = CHR$(12)
  let LFA$(5) = Text2FF$  ! USER CHOICE
```

```
end sub

sub ParseText(Source$, TextLF$, TextFF$, NumChars,HyphPct,
            WasLFFF, Fragment$)
   ! To avoid setting up LineFeed Array.
   ! This is a 'front end' to
   ! sub ParseText2( )
   dim LFA$(1)
  call SetLineFeedArray(LFA$,TextLF$,TextFF$)
  call ParseText2(Source$,LFA$,NumChars,HyphPct,
                 WasLFFF,Fragment$)
end sub

sub ParseText2(Source$, LFA$() ,NumChars, HyphPct,
             WasLFFF, Fragment$)
   ! This is the main line split control
   ! It calls for space chops and for hyphenation.
   ! How many letters are allowed at once : NumChars
   ! You send that as Source$ (can be paper) and it
   ! chews off that which fits (returned as Fragment$). It
   ! does not actually print. That offends modularity.
   ! Many kinds of programs may have specific needs
   ! for printing. One job: split.
   ! The caller to this sub may want to know if
   ! the split was by way of LF or FF code so as
   ! to handle output to screen. : WasLFFF
   ! In case an unprimed line feed array is sent,
   ! this is friendly.
            if LFA$(LBound(LFA$)) <> "LFFFSET" then
     call SetLineFeedArray(LFA$,"","")
   end if

            let Fragment$ = ""
   let LS = len(Source$)
   let WasLFFF = 0              ! Flag: user or text imbedded
                                ! LF or FF found.
   let LowestLFpos = MAXNUM     ! Flag. Initialize high lie
                                ! Probe line for LF or FF
                                ! breaks and use the most
                                ! left one (lowest).
            ! LOOK FOR A TEXT EMBEDDED LINE OR FORM CHOP
   !----------------------
   for Marker = LBound(LFA$) +1 to UBound(LFA$)
       if LFA$(Marker) <> "" then
           let LFAtpos = pos(Source$[1:NumChars],
LFA$(Marker))
           if LFAtpos > 0 then
              if LFAtpos < LowestLFpos then
                 let LowestLFpos = LFAtpos
                 let WasLFFF = Marker
              end if
           end if
       end if
   next Marker

   ! IF EMBEDDED LINE CHOP FOUND, HANDLE IT AND EXIT
   !----------------------
   if WasLFFF > 0 then
    let Fragment$ = Source$[1 : LowestLFpos -1]
    let Source$[1:LowestLFpos +len(LFA$(WasLFFF))-1] = ""
   exit sub
   end if

   ! If text does not exceed space, clean up and leave.
   !----------------------
   if LS <= NumChars or NumChars < 2 or LS < 4 then
     let Fragment$ = Source$
     let Source$ = ""
    exit sub
   end if

   ! Take space is to fill before allowing hyphenation)
   !----------------------
   if HyphPct < 0 then   ! neg val. set default
      let Hyphenate = .8
   else
      let Hyphenate = min(HyphPct,1)
   end if

   !----------------------
   ! PULL OFF ANY CODES FOR TEXT NOT TO BE FORMATTED.
   ! This may EXCEED the dimensions for printing as it
   ! is code not to be printed but parsed for program
   ! interpretation and action by the caller subroutine.
   ! The AbtrudWrapText subs handle text which might have
   ! an embedded segment ".PROMPT. whatever whatever ..!"
   !----------------------
   if Source$[1:8] = ".PROMPT." then
      ! Just one example
      ! Pull off full Prompt, of any length
      ! this list can grow
     let endpmt = pos(Source$,"!")
     let Fragment$ = Source$[1:endpmt-1]
     let Source$[1:endpmt] = ""
     exit sub
  end if

  !----------------------
  ! Make an upper case copy of the source string segment
  ! to the maximum allowed print length only.
  ! Do tests on all upper case for sanity.
  !----------------------
  let Test$ = UCASE$(Source$[1:NUMCHARS +1])

  !----------------------
  ! First Pass:
  ! Try space chop at most right portion of line as user
  ! requested. If no space to chop at then
  ! try hyphenation in same
  ! zone. We're calling that zone 'Hyphenate'.
  ! It is for both, here.
  !----------------------
  call SpaceChop(Hyphenate)
  ! Same range as hyphenation zone
  ! If a chop point was found, the source string will
  ! be smaller and the fragment, to be printed,
  ! will have something in it.
           if Fragment$ = "" and Hyphenate > 0 then
     ! Space chop failed
    call DoHyphen(Hyphenate)
  Else
     exit sub
  End if

  ! IF HERE, ABOVE FAILED. GO MORE LEFTWARD, TRY AGAIN
  !----------------------
  ! Could loop stepwise to left. I just give the zone one
  ! shot then take a leap to the left (30% from left).
  !  The .3 choice is arbitrary.
  !----------------------
  if Fragment$ = "" then call SpaceChop(.3) else exit sub
  if Fragment$ = "" then call DoHyphen(.3)  else exit sub
  if Fragment$ = "" then call SpaceChop(0)  else exit sub

  ! GIVE UP, JUST CHOP IT. (But need space for the "-")
  !----------------------
  let Fragment$ = Source$[1:numchars-1] & "-"
  let Source$[1:numchars-1] = ""
           sub SpaceChop(PctLine) ! local subsub
     ! pct -> number of chars
     let Term = int(numchars * PctLine) + 1
     for i = numchars to min(Term,numchars) step -1
         if Source$[i:i] = " " then
            ! found a space
            ! note string before the space:
            let Fragment$ = Source$[1:i-1]
                       ! delete it in source
            let Source$[1:i] = ""

            exit for
         end if
     next i
  end sub
          sub DoHyphen(PctLine) ! local subsub
   let Term = int(numchars * PctLine) + 1
   For i = numchars-1 to min(Term,numchars -1) step -1
      let j = i-1
      let PreSp = pos(Test$[i-3:i]," ") ! 321-
      let PostSp = pos(Test$[i:i+2]," ") ! -12
              ! THE FOLLOWING CODE IS FORMATTED ODDLY FOR CLARITY:
    ! actual working code is strung out
    !----------------------
    if
      (
     pos("ING-OID-TIO-RES-ERN-LIN-TAG",Test$[i:i+2]) = 0
      AND   PreSp = 0
      )
              OR (
     pos(" SUB PRE NON DIS-THAN-OVER-UNDER-UPER-POST"               Test$[i-
4:j]) = 0
       AND  PostSp = 0
       )
              OR (
     pos("BKMBNNIHCFDTFJLVXY", Test$[j:j]) = 0
      AND
     pos("GKMPQRHJCVDTFZ", Test$[i:i]) = 0
      AND
      (PreSp + PostSp = 0)
      )
            Then
```

```
      ! Hyphenate, but first
      ! handle special orphan and punctuation cases.
      ! This list may grow
      !------------------
      if Source$(j+2:j+2) = " " then
         ! Rid single char orphans
         let Fragment$ = Source$(1:j+1)
         let Source$(1:j+2) = ""
      else if pos(".,;:?!-)",Source$(j+2:j+2)) > 0 then
         ! Rid sentence end clip. Should fit
         let Fragment$ = Source$(1:j+2)
         let Source$(1:j+2) = ""
      else if pos(".,;:?!-)",Source$(j+1:j+1)) > 0 then
           ! Rid punct orphans
           let Fragment$ = Source$(1:j+1)
           let Source$(1:j+1) = ""
      else
         ! Original Algorithm
         let Fragment$ = Source$(1:j) & "-"
         let Source$(1:j) = ""
      end if
                 exit for
   end if
   next i
 end sub
end sub
```

Now that the line is clipped to fit neatly, you might justify it within the desired print space. You may decide only to allow 10 letters in a space that might be able to hold 13 (margins etc.). Keep clip and justify apart.

```
        !-------------
!! JUSTIFY TEXT IN GRAPHIC CONTEXT ------
!-----------
! Consider:
! *  spacetofill with text  *
! |---------------------| <- here-> 24 chars
! 8: : : -SpaceToFill- - : :81
        ! Page
!      9 : Left edge and allowed span (SpaceToFill)   or
!      8 = Left edge and 81 = right edge              or
!      8 = Left edge and Chars to allow on a line.
! All 3 define the same space.
! Several slave subs allow the main sub to be called according to
! program ease of variable handling.

sub PlotJustifiedFastUp(Strg$,X,Y,YOff, Cbw, SpaceToFill,
                        JustPct, Margins)
   ! JustPct: <0 = default (line of 60% text,) to 100%
   !                where text means packed nonspace
   !                characters.)
   !            0 = no justification
   !        0<=1 = pads % of line text to -> justification
   !          >1 = num of end spaces to -> justification
   ! Margins are stated in number of characters
   !          (.5,1,2 etc.) L & R
   ! YOff generally = 0. If not, text is plotted twice
   ! with this offset
   ! such as 1 pixel to reduce flicker in interlace.
   ! Strg$ the passed string.
   ! Plot text starting at X,Y
   ! Plot it again, maybe, offset vertically by
   ! YOff to kill flicker.
   ! Tell this sub what the character width is so it need
   ! not refigure over and over if called in a loop.
   ! SpaceToFill - graphic terms, the space avail to print
   ! Margins: as chars
   ! JustPct, ? justify at all or
   ! just when to do it or its low.
            dim A$(90) ! up to 90 'words' to a line
            ! A  'word' can be 1 char.
            ! Work with a copy, trim off trailing space.
   let String$ = RTrim$(Strg$)
   let LS = len(String$)
            ! Treat left space as firm
   ! It is an indent not to be justified.
   let B$ = LTrim$(String$)
            let Indent = (LS - len(B$)) * Cbw
   let ACbw = abs(Cbw) ! Cbw can be negative in True BASIC
   let ASpaceToFill = abs(SpaceToFill) ! left can be
                                        ! > than right
            if Margins <= 0 then
     if Margins > 0 then
      ! Clip off margin space first.
      let ASpaceToFill = ASpaceToFill -2 * ACbw * Margins
```

To do or not to do—that is the question.

Roy M. Nuzzo

True BASIC has several very off-beat and useful extensions.

1. There are scripts which behave like batch files within the True BASIC command window.

2. There is the extended use of direct command mode which allows use of libraries directly, without need for any programming. This can be used as the ultimate desktop calculator. But how do you keep a written record of it?

3. There is the "ECHO" function which acts as a stenographer for the command window and can divert a copy of all command window text to a file or to the printer. A paper trail.

4. There are "Do Programs" which act in the background and have the ability to look at and react to and/or alter programs currently in the editor or do whatever else a program might be asked to do.

This piece will familiarize you with the last two.

Scripts are just text files of commands as they might be typed directly into the command window. They are acted on as if they were directly typed by you. If you want to keep a record of this beyond the running shell-like list of old actions kept automatically by the command window), then just type "ECHO".

ECHO defaults to the printer. Anything typed or printed to the command window also appears at the printer. "ECHO TO filename" creates a text file which is a file version of the printer output without the printer output. Therefore, if you do a series of commands relating to a complex calculation, all of it is recorded for you.

"ECHO OFF" shuts the echo function off.

Scripts can turn ECHO on and off, as they can perform any command.

"Do programs" are very handy and interesting. Any library can be made to act as a "do program" or better yet be called by a do. The 'do' has only one iron-clad rule. The argument list of the very first sub (must be a sub) must be a single dimension string array followed by a string:

```
EXTERNAL  ! <— indicates a library without program code to be
executed
   sub MyDo(line$(),String$)
   ...code
end sub
      other subs  and functions may follow.
```

The 'do' sub, the first one in the library, is activated when the library name is evoked by a 'do' command. If the above library were named 'Glitz', then a command 'do Glitz' would automatically activate the sub MyDo() as a background task. Notice that the sub name is irrelevant. It is simply the first sub and it has the needed two arguments. Typically, you write this sub as if it were a program. The 'end sub' statement is the 'end' statement.

```
      let LM = ChW * Margins  ! set new X to plot text
                              ! at. left margin.
   else
     ! default = 1 char
     let LM = ChW
   end if
end if
let XN = X + Indent + LM
          Select Case JustPct
   case is = 0 ! Use default
      ! let SpToJust = 2 +( 1( * (TotalChars) )
        let SpToJust = 2 +(.15 * (ASpaceToFill / AChw))
        let Trigger = ASpaceToFill - AChw * SpToJust
             case 0      ! Justify is off. Do as is.
        plot text, at X + LM, Y : String$
        if YOFF <> 0 then
           plot text, at  X +LM, Y +YOFF : String$
        end if
        exit sub
             case is <> 0 ! # of Chars at end to => justify
        let Trigger = ASpaceToFill - AChw * JustPct
             case else
      let Trigger = JustPct * ASpaceToFill
End Select
          do
     let p = pos(S$," ")
     let words = words + 1
     if p > 0 then
        ! COLLECT WORDS-- AN ARRAY WITH SPACES TRIMMED
        ! BASED ON SPACES BETWEEN WORDS
        let A$(words) = Trim$(S$[1:p-1])
        let S$ = Trim$(S$[p+1:MAXNUM])
     else
        let A$(words) = S$
     end if
              ! TALLY TOTAL CHARACTERS MINUS THE SPACE BETWEEN
     ! THIS IS A TOTALLY EYE ORIENTED SCHEME, EMPIRIC
    let NonSpChars = NonSpChars + len(A$(words))
loop while p > 0
          let ACharFill = NonSpChars * AChw  +  abs(Indent)
          if ACharFill >= Trigger then
   !————————————————————
```

```
      ! If the solid area (nonspace) exceeds the user
      ! requested trigger then go ahead and expand the
      ! space between words, by pixels:
      ! |<-- --- --- -- --- ----->|<———————trigger|
      ! (Hi my name is Fred What is your name?               |
      ! |<-         ACharFill   ->|<- TotalSpace  ->|
      ! (Hi  my  name  is  Fred   What  is  your  name?|
      !——————————————————————
                 ! Where TotalSpace is total space to be used up by
      ! word spacing, figure space needed between each
      ! word to expand to right edge. For ease, use the
      ! absolute val of the print zone length.        !——————————————————

      let TotalSpace = ASpaceToFill - ACharFill
                ! Don't justify 2 words, never looks right
      if words > 2 then
         !——————————————————
         ! Set the space polarity to that of the sign of
         ! the char width.
         ! Each space = total space divided by word            ! intervals.
         ! Remember that the interval count between 5
         ! fingers is 4.
         !——————————————————                  let EachSpace = sgn(ChW) *
(TotalSpace/(words-1))
      else
         let EachSpace = ChW   ! normal single character
      end if
               !——————————————————
      ! Plot one word at a time and add its following
      ! space, then move the left plot point to that spot
      ! and do next word. The first word start is figured
      ! above. Just move it along.
      !——————————————————
      for w = 1 to words
         plot text, at XN, Y : A$(w)
         if YOff <> 0 then
            plot text, at XN, Y + YOFF : A$(w)
         end if
                  let XN = XN + len(A$(w)) * ChW +EachSpace
         !    ^^plots  ^^width of word   ^^width of space
```

(continued on page 74)

When the sub is exited, the editor becomes active once again (it freezes during the action of the 'do' program.

You pass one argument to this sub. If you supply none, then a null is automatically passed. You do not pass anything to the array. This array is supplied by the 'do' mechanism for you. Therefore, if you typed, from the command window, "do Glitz, "Hello Fred."", then the String$ argument would carry "Hello Fred.". As far as the call is concerned, there is only this one argument, the simple string argument.

That one string can be huge and carry all sorts of numbers and string and character values within it, as True BASIC has very refined tools for loading and unloading complex data from strings. You system programmers will note that strings are used to form 'structures' when they are needed and this means that they can be Window structures, ViewPort structures, you name it structures, whatever.

The line$() array is automatically passed. Its lower limit is always 1, and its upper limit is equal to the number of lines of code currently in the editor. Line$(1) carries the first line of code, Line$(2) the second etc. This is not a copy of the code in the editor, it *is* the code. If you do neat things to this code, you do neat things to the actual program in the editor.

Like what?—like remove or alter remarks, or look for special code within remarks to cause a file to be created (auto docs) that update programming documentation. Or, alter spacing and capitalization to fit a standard for style of the code. Or, strip CHR$(13) from the code (load text files from IBM to Amiga, you may need to get rid of the extra CHR$(13)s from the IBM world). Or - strip out tab characters and replace each with three spaces (hello 'z' C edit people). Or - use as a preprocessor swapping tokenized code for expanded code. Here, the algorithms for the swap can get as hairy, and as dangerous, as you can stand.

Here are a few examples:

```
! 'Strip 13 Do'
! This file is saved as 'Strip_13_Do' for source file
! but in compiled form as 'Strip13' in the TBDo drawer.
! To use, type 'do Strip13' from the command line or click on it
! from the menu requester for 'do' files.
! The file in the editor will have all CHR$(13)s removed and the tokens
! that indicate their presence will disappear. Resave if you want.
        EXTERNAL
        SUB Strip13_DO(line$(), arg$) ! will ignore arg$, irrelevant
  declare def Before$, After$, SwapChar$
            for i = 1 to UBound(line$)
     if pos(line$(i),CHR$(13)) > 0 then
        let line$(i) = SwapChar$(line$(i),CHR$(13),"")
     end if
  next i

 DEF Before$(a$,w$) ! COPY OF ALL CHARS BEFORE w$
     let p = pos(a$,w$)
     if p = 0 then
        let Before$ = ""
     else
        let Before$ = a$[1:p -1]
     end if
 end def

 DEF After$(a$,w$) ! COPY OF ALL CHARS AFTER w$
     ! 'MAXNUM' is a True BASIC constant which returns
     ! the largest number that the computer you running
     ! can handle legally. Good upper limit for strings.
                  let p = pos(a$,w$)
     let L = len(w$)
     if p = 0 then
        let After$ = "" else let After$ = a$[p+L:Maxnum]
     end if
 end def
```

(continued on page 74)

Assembly Language & Computer Simulations

by Bill Nee

Using the Amiga 500 and now my newly acquired Amiga 3000 for computer simulations is one of my favorite pastimes. The speed at which these machines run, coupled with 32 colors, make for very interesting displays. And with the computer you can easily change values to see what effect new parameters have. In this article we'll do a simulation showing how a virus can spread between cells. Values to be used will be passed from CLI/SHELL with the program name, and I'll show you yet another PSET routine.

The Simulation

The simulation we're using was first described by A. K. Dewdney in *Scientific American* (8, 1988). An array is filled with cells on a random basis with values from 0 to 50. Based on the relationships described below, new cell values are computed, stored in a second array, then transferred back to the first array. Cells in the first array are colored according to a color scheme within the program.

Each cell is considered to be in one of three states depending on its value. A cell with a value of 0 is HEALTHY, a cell with a value between 1 and 49 is ILL, and a cell with a value of 50 is DEAD. A dead cell will become a healthy cell at the next generation. That's the easy one. To discuss the next two states we need to learn a few variables:

AA - number of ill cells, not less than 1
BB - number of dead cells
K1 - a weighting parameter (2 is the default value)
K2 - a weighting parameter (3 is the default value)
GG - the infection constant (use 1-20; 6 is the default value)
S - sum of a cell's value plus its four neighbor values

The formula for the next generation of a healthy cell (value 0) is INT(AA/K1)+INT(BB/K2). Generally, keep K1 and K2 between 1 and 4; of course, don't let one of them ever be 0! The formula for the next generation of an ill cell is INT(S/AA)+GG; if this value, however, exceeds 50 the new value will be 50. The program will wrap-around so the right neighbor of a cell on the right side is actually the cell on the left side and vice-versa; the same applies to the top and bottom. With these three rules you can create patterns of continuously changing color; some never settle down while others become ever growing spirals.

Listing 1 is the program for this simulation. Since the test of each cell and the computation of the next generation cell is the most important routine, I made this a macro near the start of the program. You could have it as a subroutine instead, but then the current program address must be saved each time and returned; since we've got the space, I used it as a macro. Because TEST is the heart of the program, I'll go through it in detail.

The Macro

The four values passed to TEST are the locations of each of the four neighbors (above, left, right, and below) in relation to the current cell; the value of the current cell is in d0. Since MOVE affects the zero flag we can check right away and see if the current value is 0; if so, branch to HEALTHY. If it's not 0, compare the value to 50; if it's not equal, branch to ILL. The only choice left is that the value must be 50, so store the next generation's cell value of 0 in d0 and branch to TESTDONE.

If a cell is HEALTHY, first put a 1 in AA (its value is never less then 1) and a 0 in BB. Next, move a neighbor's cell value into d1. If this value is 0, go immediately to the next neighbor check since a zero won't affect anything. However, if the neighbor's cell value is 50, increase BB by 1; else increase AA by 1. After all four neighbors have been checked, divide AA by K1 and BB by K2 then add AA and BB. This new value will be the cell value for the next generation.

The final test is for an ILL cell. Again, put a 1 in AA; we won't be using BB this time. Get a neighbor's cell value in d1. If it's 0, branch to the next neighbor check; if not, add the neighbor's cell value to the current cell value. If the neighbor's cell value is 50, branch to the next neighbor check, else increase AA by 1. Go through all four neighbors in the same manner. When you've finished, d0 contains the neighborhood's sum and AA is the number of all ill cells, then divide d0 by AA. Here's where you have to have planned ahead.

I kept the maximum cell value at 50 since, at this point, the neighborhood's sum in d0 could be 49+4*50 or 249. While you could squeak by using 51, larger maximum values could result in a d0 value greater than one byte. Next you would add GG to the division result and compare the total to 50. But if the result in d0 was, for example, 245 and you added a GG of 15 the byte value in d0 would be 5 and this would appear to be less than 50 when it's actually 260. So I first check the division result against the difference between 50 and GG. If it's greater than this, adding GG will make the result greater than 50 so put 50 in d0 as the next generation's cell value. If the value is not greater than the difference you can safely add GG and use this as the new next generation's cell value.

The Listing

Now let's look at Listing 1 in more detail. Since they're used so often, I equated AA and BB to registers d3 and d4 as well as equating SUM, ACROSS, and DOWN to their registers. The length of the array must be a multiple of 32; more about that later when I discuss the new PSET routine. Several variables are equated to the length (LENM1, LEMP1, etc.) as well as the size of the array needed. I'll explain the other variables when we get to PSET. Next, there are several macros as well as the TEST macro.

The program starts in standard fashion opening the Intuition and Graphics libraries as well as a 320x200 16-color screen and window.

►

TABLE I

K1	K2	GG	K1	K2	GG	K1	K2	GG
1	4	6	2	2	10	3	1	7
1	3	5	3	1	5	4	1	3
3	1	15	1	1	10	2	3	6

The RANDOM routine uses the CIA register to fill Array1 with random values between 0 and 50. The next part of the program reads the values you passed, if any, along with the program name. Enter values for K1, K2, and GG separated by a space or comma: for example, 'SICKWELL 2 3 6' or 'SICKWELL 3,2,18'. If you don't enter any values the program will default to 2,3,5 respectively.

When you enter values in this way at the start of the program a0 contains the location of these values and d0 contains the number of characters entered. The first two values must be one digit each while the last value could be one or two digits. I haven't included any checks for incorrect values; I'll leave that up to you. You might also want to add a routine allowing for a "?" instead of a value and then have a message printed reminding you of what's to be entered along with the acceptable ranges.

Now the program must check every cell to compute its next generation value. To accomplish the wrap-around there is a separate routine for each of the four corners, top row, bottom row, sides, and center square. If you use a length of 32*5 or 160 the top row actually goes from 0 to 159, the next cell in the array is 160. The four neighbors of the upper-left cell (cell 0) are located at LEN*LENM1, LENM1, 1, and LEN (top, left, right, and bottom) away from cell 0. The neighbors for the top row are LEN*LENM1, -1, +1, and LEN away from each cell. The neighbors for the upper-right cell (cell LENM1) are LEN*LENM1, -1, -LENM1, and LEN. For the center square starting at cell LENP1 the four neighbors are -LEN, -1, +1, and LEN away. It may be easier to draw a box, divide it into smaller squares and label some of the cells to see the relationship. Just remember that the upper-right square is LENM1.

Another PSET Routine

When all of the new cell values are in Array2, it's time to transfer them back to Array1 and PSET them. I said that I'd discuss another PSET routine so here goes. This routine is based on the "Fast Fractals" article in *Amazing Computing* (V 4, 11) and a program sent to me by Stan Jurgielewicz of Virginia. Stan is a real renaissance man and an expert at just about everything. He doesn't hesitate to rewrite my programs in five different ways showing me better methods to do everything.

This routine fills data registers with 32 values at a time and actually pokes them into the proper screen locations. That's why the arrays must be multiples of 32 across. Different data registers are used to hold each of the four bits that make up each color. Since the bitplanes are an equal distance apart ($1F40 bytes) we only need the location of the first one. SUM contains the current color. Rotating it to the right moves the first color bit into the X bit of the status register; rotating a data register with X carry (ROXL.L) brings the same bit into the data register. If there are four color bits, you must use four data registers. Do this 32 times and each of the four data registers contains one long word of the color value for one bitplane.

Now we have to poke these words into a screen location so we can see them, but where? When we PSET the random values each point was at an XOFF ((320-LEN)/2) and YOFF ((200-LEN)/2) to center the picture. So let's compute the byte that contains these two offsets. There are 40 bytes in each horizontal line (320/8), so the first byte in the row we want is YOFF*40. The XOFF is how many more bits in we go, and since there are 8 bits per byte, divide the XOFF by 8. Add the two values together and that's the start of the byte we need. I defined this at the start of the program using BYTE EQU (YOFF*320+XOFF)/8. To access this location put bitplane1 address in a1 and offset it by byte (LEA BYTE(A1),A1).

To get the color, MOVE the entire first bit register into a1, the second bit register into a1+$1F40, the third bit register into a1+$1F40*2, etc. Actually, do this in reverse so you can end with "(A1)+", automatically increasing a1 by a long word. How many times do you do this across? I use the variable WPL (long Words Per Line) which is just LEN/32. This is the number of times we'll poke color into the screen location going across.

When you finish a line, where is the next byte? A full-screen display would continue on with the next byte but we need to move enough to reach the end of the screen and go on to the next row's X offset. This distance is 2*XOFF in bits, or 2*XOFF/8 bytes. I call this variable BYTEOFF. Using these variables and always making the length a multiple of 32 lets you automatically display different size arrays without re-computing all of the variables. Keep the length at 5*32 or smaller. You must re-assemble the program to change the length.

Since state zero is so important for cells in the array it gets its own color; any other state value is divided by four and increased by one to get its color. I wrote this program on an Amiga 500 having it switch array values before showing the new color. This cuts down on the slight flicker as the new colors are PSET. For faster computers you can eliminate the SWITCH routine and combine it with the new color PSET routine. The program keeps running until you press the LMB. After typing in this program save it as SICKWELL.ASM; assemble it with A68K and BLINK it using SICKWELL.O. Run the program as SICKWELL [K1 K2 GG]. For your convenience, I've included A68K, BLINK, their docs and the assembled version of SICKWELL on the magazine disk.

Table I is a list of some interesting combinations for K1, K2, and GG. Try your own combinations; running the same combination several times in a row may produce different results each time due to the random distribution of initial values.

Listing

```
;LISTING1
equates:
depth         = $4
jam2          = $1
mousebuttons  = $8
borderless    = $800
wa_screen     = $2e
wa_rport      = $32
nw_screen     = $1e
customscreen  = $f
activate      = $1000
rmb           = $10000
public        = $1
chip          = $2
fast          = $4
clear         = $10000
```

WOMEN GIVES BIRTH TO 400 POUND BOG MONSTER!

AMIGA

A4000 Computer	2599
A2000 Computer	599
A1200 Computer	599
A600 Computer	329
1960 Multisync Monitor	499
2024 Monochrome Mon.	199
A590 20MB HD System	239
A570 CD-ROM	415
CDTV	574
1084S Monitor	279
A2386 25Mz Bridgecard	499
A2320 Flicker Fixer	145
A2232 Multi-Serial card	245
A520 Video Adapter	29
A2088 XT Bridgecard	99
A2090A Hard Drive Controller w/45MB HD	149
HD Floppy Drive 1.76MB	99.95

CBM CHIPS

Kickstart 2.1 Upgrade Kit	85
Kickstart 1.3	26
1MB Agnus (8372A)	Call
2MB Agnus (8372B)	79.95
Super Denise (8373)	33
Paula (8364)	22
CIA (8520)	9.95
Gary (5719)	14.95
2620/2630 Eprom Kit for CBM Accelerator	35

IVS

Grand Slam	229
Grand Slam 500	267
Trumpcard Pro	139
Trumpcard 500 Pro	225
Trumpcard 500 Plus	Call
Trumpcard 500 AT	164
Meta4 Memory Card	85
Printerface Auxiliary Printer Port	55
Sourcer Switching Power Supply	99

GVP

A500-HD8+0MB/40	349
A500-HD8+0MB/80	425
A500-HD8+0MB/120	479
A500-HD8+0MB/213	599
A530-HD8+1/120	759
A530-HD8+1/245	Call
A2000-HC8+0MB	149
SIMM32/1MB/60ns	59.95
SIMM32/4MB/60ns	184
1MB SIMM GForce A3000	Call
G-Lock Genlock	399
A3000-Impact Vision 24	1675
A2000-IV24 Adapter	39
VIU-CT	499

A1230 Turbo+
A1200 SCSI / RAM+

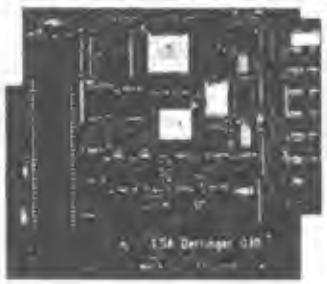
GForce Accelerators w/ 68030,68882, 40ns RAM, SCSI Controller, RAM Card in One

25Mhz/1MB	525
40Mhz/4MB	695
50Mhz/4MB	1195
68040 33Mhz A2000/4MB	1445
68040 28Mhz A3000/2MB (40ns RAM)	975

PC286 Module 16Mhz	109
Tahiti-II 1GB (35ms)	3300
Tahiti-II 1GB Cartridge	299
Syquest 44MB Removable	275
44MB Cartridge	65
Syquest 88MB Removable	385
88MB Cartridge	119
Impact XC Ext. case	250
FaaastROM Kit	35
Cinemorph Software	109
Phonepak VFX	379
DSS8 Sound Sampler	74
I/O Extender	195
Image F/X	239

CSA DERRINGER

Running at 25Mhz w/ MMU
4MB 32bit RAM Exp. to 32MB,
w/ 68881 $499
w/ 68882 add $75
w/ 68882-50Mhz add $150
8MB version add $199.

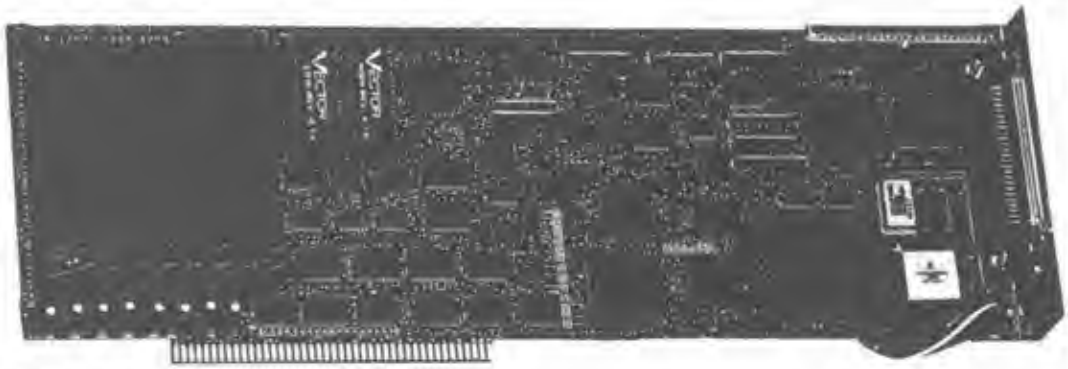


ASK ABOUT OUR ACCELERATOR, HARD DRIVE AND MEMORY UPGRADES

SLINGSHOT
SINGLE A2000 SLOT FOR THE A500
$39.95

KOOL-IT
MICRO FAN COOLING KIT FOR THE A500
$39.95

QWIKA SWITCHA
4 SOCKETED ROM SELECTOR
$39.95



VECTOR
BY INTERACTIVE VIDEO SYSTEMS

THIRD GENERATION 68030 PROCESSOR ACCELERATOR FOR THE AMIGA 2000

TESTED & CLOCKED AT 25 MHZ - W/68030 & 68882, EXPANDABLE TO 32MB RAM W/ SCSI CONTROLLER, FEATURES PROPLEX, SCSI SHARE... NETWORKING, RAM & SCSI USABLE IN 68000 MODE.

Call!

HARD DRIVES

Quantum - Conner
Seagate - Fujitsu
Western Digital
2.5" and 3.5"
At unbelievable prices!

US ROBOTICS

16.8K Courier HST with fax	595
16.8K Courier HST Dual Standard with fax	925
Courier v.32bis	449

A2630 Accelerators

68030/68882 25Mhz
4MB 32bit RAM
Newest Eproms
$499

MEMORY CHIPS

IVS 1MB SIMMS	29.95
1x8 80-60ns SIMMS	Call
4x8 80-60ns SIMMS	Call
1x4 80-60ns Static ZIP	Call
1x4 80-60ns Page ZIP	Call
1x4 80-60ns Page DIP	Call
1x1 100-70ns DIP	Call
256X4 80-60ns DIP	Call
256X4 80-60ns ZIP	Call
A4000 SIMMS	Call

A600/1200 Accesories	
PCMCIA 2 and 4MB	Call
MBX1200	Call
Baseboard	Call

LASER PRINTER MEMORY

HP II, IID, IIP, III, IIID, IIP AND ALL PLUS SERIES	
Board with 2MB	87.50
Board with 4MB	145
Deskjet 256K Upgrade	89.95
2 Boards (for 500 Series)	130

DKB

Insider II w/0K	159
2632 w/4Megabytes	399
MegAChip 2000/500 w/2MB Agnus	269
Multi-Start 2 Rev 6A	53
KwikStart II for A1000	89
SecurKey Security Board	99
BattDisk battery backed static RAM disk	199

ACCESORIES/MISC.

Kick Back ROM Switcher	35
PowerPlayers Joystick	8.50
CSA Rocket Launcher	499
SupraTurbo 28Mhz	Call
68030 50Mhz CPU	199
68882 50Mhz Math Co	149
Safeskin Protectors	Call
Xtractor+ Chip Puller	9.95

Maxtor 213MB
15ms LPS
6K Cache
$375

DeVine COMPUTER SALES

18 Wellington Drive
Newark, DE. 19702
(800) 578-7617 ORDERS ONLY
(302) 836-4138 Info 10AM-6PM
(302) 836-8829 Fax 24 HOURS

Visa / Master Card Accepted. Prices And Specifications Are Subject To Change Without Notice 15% Restocking Fee On All Non-Defective Returned Merchandise. Call For Approval RMA# Before Returning Merchandise. Shipping And Handling For Chips Is $4 COD Fee $5 Personal Checks Require 10 Days To Clear. Call For Actual Shipping Prices On All Other Items. **If You Don't See It Here, Call Us!**

```
offsets:
;exec
openlibrary   = -552
closelibrary  = -414
allocmem      = -198
freemem       = -210
forbid        = -132      ;no registers used
permit        = -138      ;no registers used
;intuition
openscreen    = -198
closescreen   = -66
openwindow    = -204
closewindow   = -72
ViewportAddress = -300  ;a0=window
;graphics
setdrmd       = -354
loadrgb4      = -192    ;a0=vp,a1=colortable,d0=#pens
setapen       = -342
writepixel    = -324

;dn     equr  d7
across  equr  d6
down    equr  d5
bb      equr  d4
aa      equr  d3
len     = 32*8
lenm1   = len-1
mlenm1  = -1*lenm1
lenm2   = len-2
xoff    = (320-len)/2     ;to center display
yoff    = (200-len)/2     ;to center display
lenp1   = len+1
lenm3   = len-3
mlen    = -1*len
byte    = (yoff*320+xoff)/8 ;byte containing yoff/xoff
wpl     = lenm/32+1       ;long words per line
Bytesoff = 2*xoff/8       ;offset to next byte
Siz6    = len*len         ;array size
;R1     equ  2            ;default
;R2     equ  3            ;default
;R6     equ  6            ;default

macros:
syslib  macro             ;(routine)
  movea.l  4,a6
  jsr     \1(a6)
  endm

openlib macro             ;(name,location,#RO if 0)
  lea     \1,a1
  moveq   #0,d0
  syslib  openlibrary
  move.l  d0,\2
  beq     \3
  endm

intlib  macro             ;(routine)
  movea.l intbase,a6
  jsr     \1(a6)
  endm

openscreen macro          ;(parameters,screen,#RO if 0)
  lea     \1,a0
  intlib  openscreen
  move.l  d0,\2
  beq     \3
  endm

openwindow macro          ;(parameters,window,#RO if 0)
  lea     \1,a0
  move.l  screen,nw.screen(a0)
  intlib  openwindow
  move.l  d0,\2
  beq     \3
  endm

gfxlib  macro             ;(routine)
  movea.l gfxbase,a6
  jsr     \1(a6)
  endm

test  macro               ;(top,left,right,bottom)
  beq.s   healthy\@
  cmpi.b  #50,d0
  bcs.s   ll\@            ;branch if lower
  moveq   #0,d0
  bra     testdone\@
healthy\@
  moveq.b #1,aa           ;always at least 1
  moveq.b #0,bb           ;start bb at 0
h1\@
  move.b  \1(a4),d1       ;cell 'above' value
  beq.s   h2\@            ;branch if 0
  cmpi.b  #50,d1          ;is it 50?
  beq.s   h1a\@
  addq.b  #1,aa           ;if not, increase aa
  bra.s   h2\@
h1a\@
  addq.b  #1,bb           ;increase bb
h2\@
  move.b  \2(a4),d1       ;'left' neighbor value
  beq.s   h3\@
  cmpi.b  #50,d1
  beq.s   h2a\@
  addq.b  #1,aa
  bra.s   h3\@
h2a\@
  addq.b  #1,bb
h3\@
  move.b  \3(a4),d1       ;'right' neighbor value
  beq.s   h4\@
  cmpi.b  #50,d1
  beq.s   h1a\@
```

```
  addq.b  #1,aa
  bra.s   b4\@
b3a\@
  addq.b  #1,bb
b4\@
  move.b  \4(a4),d1     ;'bottom' neighbor value
  beq.s   b5\@
  cmpi.b  #50,d1
  beq.s   b4a\@
  addq.b  #1,aa
  bra.s   b6\@
b4a\@
  addq.b  #1,bb
b5\@
  moveq   #0,d1
  move.b  k1,d1
  divu    d1,aa         ;aa=aa/k1
  moveq   #0,d1
  move.b  k2,d1
  divu    d1,bb         ;bb=bb/k2
  add.b   bb,aa         ;aa=aa+bb
  move.b  aa,d0         ;new generation value
  bra.s   testdone\@
i1l\@
  moveq.b #1,aa         ;aa always at least 1
  move.b  \1(a4),d1     ;'top' neighbor value
  beq.s   i112\@        ;branch if 0
  add.b   d1,d0         ;add it to current value
  cmpi.b  #50,d1        ;is it50?
  beq.s   i112\@
  addq.b  #1,aa         ;increase aa by 1
i11j\@
  move.b  \2(a4),d1
  beq.s   i113\@
  add.b   d1,d0
  cmpi.b  #50,d1
  beq.s   i113\@
  addq.b  #1,aa
i113\@
  move.b  \3(a4),d1
  beq.s   i114\@
  add.b   d1,d0
  cmpi.b  #50,d1
  beq.s   i114\@
  addq.b  #1,aa
i114\@
  move.b  \5(a4),d1
  beq.s   i115\@
  add.b   d1,d0
  cmpi.b  #50,d1
  beq.s   i115\@
  addq.b  #1,aa
i115\@
  divu    aa,d0         ;total values/aa
  cmp.b   diff,d0       ;greater than 50-gg?
  bgt.s   i116\@
  add.b   gg,d0         ;ok to add gg
  bra.s   testdone\@
i116\@
  moveq.b #50,d0        ;can't exceed 50
  bra.s   testdone\@
dead\@
  moveq   #0,d0         ;dead cell becomes healthy
testdone\@
  move.b  d0,(a5)+      ;save new value;increase array
  endm

pset macro    \.
     move.l   rp,a1
     move.w   across,d0
     add.w    #xoff,d0    ;center across
     move.w   down,d1
     add.w    #yoff,d1    ;center down
     ext.l    d0
     ext.l    d1
     gfxlib   writepixel
     endm

color macro
     move.l   rp,a1
     gfxlib   setapen
     endm

array macro             ;address,beg=0|1
  move.l  #size,d0
  move.l  #510004,d1
  syslib  allocmem
  move.l  d0,\1
  beq     \2
  endm

eveppc macro
     ds.w    0
     endm

start:
  move.l  sp,stack
  movem.l a0/d0-\(sp)   ;save lot parameters
address,beg=0
open_libs:
  openlib intuition,intbase,done
  openlib gra/fx,gfxbase,close_int

ser_up:
make_screen:
  openscreen myscreen,screen,close_all
  openwindow mywindow,window,close_screen
  move.l  d0,a0
  movea.l ww.sport(a0),a1
  move.l  a1,rp
  movea.l window,a1
  intlib  viewportaddress
  move.l  d0,vp         ;viewport address
```

```
    movea.l d0,a0
    lea     color_table,a1
    moveq   #16,d0          ;16 new colors
    gfxlib  LoadRGB4

    movea.l rp,a1
    move.l  #pens,d0
    gfxlib  SetDrMd
get_bit_planes:
    movea.l rp,a0
    movea.l 4(a0),a0
    lea     bpl,a0
    move.l  8(a0),(a0)+     ;bitplane addresses
    move.l  12(a0),(a0)+
    move.l  16(a0),(a0)+
    move.l  20(a0),(a0)+
    move.l  24(a0),(a0)

memory:
    array   array1,close_window
    movea.l d0,a4
    array   array2,close_out

random:
    nop
1   move.b  $bfe801,d7      ;CIA register
    movea.l a0,a1
    movea.l gfxbase,a0
    moveq   #0,down
r1  move.b  $bfe901,d4      ;CIA register
    nop
    moveq   #0,across
r2  move.b  $bfe801,d4
    nop
    mulu    d7,d4
    add.b   across,d4
    cmpi.b  #50,d4          ;maximum cell value
    bhi.s   r2
    move.b  d4,d3
    move.b  d4,num
    cmpi.b  #0,d0           ;0 gets its own value
    beq.s   nocolor
    ext.b   #2,d0           ;color 2
    addq.b  #1,d0
nocolor
    color
    pset
    move.b  num,(a4)+
    swap    d4
    addq.w  #1,across
    cmp.w   #len,across
    bne.s   r2
    addq.w  #1,down
    cmp.w   #len,down
    bne.s   r1
    gfxlib  OwnBlit
```

```
get_cli_input
    movem.l (sp)+,a0/d0     ;parameter location
    movea.l d0,a0
    move.l  d0,d7           ;# of characters
    subq    #1,d7
    beq.s   default         ;no parameters
    move.b  (a0)+,d0        ;get first value
    subi.b  #$30,d0         ; and normalize it
    lea     1(a0),a0        ;skip space or comma
    subq    #2,d7           ;2 less characters to read
    move.b  (a0)+,d1        ;get second value
    subi.b  #$30,d1         ; and normalize it
    lea     1(a0),a0        ;skip space or comma
    subq    #2,d7           ;2 less characters
    move.b  (a0)+,d2        ;get last value
    subi.b  #$30,d2         ; and normalize
    subq    #1,d7           ;last character?
    beq.s   clidone         ;branch if so
    move.w  #10,d1          ;that was 10's digit
    and.w   #$ff,d2         ;clear rest of word
    mulu.w  d1,d2           ;times 10
    add.b   (a0),d2         ;add 1's digit
    subi.b  #$30,d2         ; and normalize
    bra.s   clidone
default:                    ;use default values
    move.b  #2,d0           ;#1
    move.b  #3,d1           ;#2
    move.b  #5,d2           ;#3
clidone:
    move.b  d0,k1
    move.b  d1,k2
    move.b  d2,gg
    moveq.b #55,d0
    sub.b   gg,d0
    move.b  d0,diff         ;50-gg
    lea.l   bpl,a0

newgen:
uleft                       ;upper-left cell 1
    movea.l array1,a4
    movea.l array2,a5
    moveq   #0,d3
    move.b  (a4),d0         ;its value
    test    len*len-1,len-1,1,len

trow                        ;cells 1 -- (len-1)
    movea.l array1,a4
    lea     1(a4),a4        ;cell 1
    movea.l array2,a5
    lea     1(a5),a5
    moveq   #1,across
    move.w  #(len-2),across ;number of times
t1
    moveq   #0,d3
    move.b  (a4),d0
    test    len*len-1,-1,1,len
    lea     1(a4),a4
```

```
  dbf     across,t1

uright                  ;upper-right,cell lenmth-1
  movea.l array1,a4
   lea     lenm1(a4),a4
  movea.l array2,a5
   lea     lenm1(a5),a5
   moveq   #0,d0
   move.b  (a4),d0
  test    len*lenm1,-1,mlenm1,len

leftside
  movea.l array1,a4
   lea     len(a4),a4
  movea.l array2,a5
   lea     len(a5),a5
   moveq   #0,down
   move.w  #lenm2,down
ls1
   moveq   #0,d0
   move.b  (a4),d0
   test    mlen,lenm1,1,len
   lea     lenm1(a4),a4   ;move to
   lea     lenm1(a5),a5   ; rightside
   moveq   #0,d0
   move.b  (a4),d0
   test    mlen,-1,mlenm1,len
    lea     1(a4),a4
    dbf     down,ls1

lleft                       ;lower-left
  movea.l array1,a4
   lea     len*lenm1(a4),a4
  movea.l array2,a5
   lea     len*lenm1(a5),a5
   moveq   #0,d0
   move.b  (a4),d0
  test    mlen,lenm1,1,len*mlenm1

brow                        ;bottom row
  movea.l array1,a4
   lea     (len*lenm1+1)(a4),a4
  movea.l array2,a5
   lea     (len*lenm1+1)(a5),a5
   moveq   #0,across
   move.w  #lenm2,across
brow1
   moveq   #0,d0
   move.b  (a4),d0
   test    mlen,-1,1,len*mlenm1
   lea     1(a4),a4
   dbf     across,brow1

lright                      ;lower-right
  movea.l array1,a4
   lea     lenm1*lenp1(a4),a4
  movea.l array2,a5
   lea     lenm1*lenp1(a5),a5
  moveq   #0,d0
  move.b  (a4),d0
  test    mlen,-1,mlenm1,len*mlenm1

center                      ;center square
  movea.l array1,a4
   lea     lenp1(a4),a4
  movea.l array2,a5
   lea     lenp1(a5),a5
   moveq   #0,down
   moveq   #0,across
  move.w  #lenm2,down
l1
  move.w  #lenm2,across
c1
   moveq   #0,d0
   move.b  (a4),d0
   test    mlen,-1,1,len
   lea     1(a4),a4
   dbf     across,c1
   lea     2(a4),a4
   lea     2(a5),a5
   dbf     down,l1

switch                      ;for slower computers
  movea.l array1,a4
  movea.l array2,a5
  move.w  #lenm1,down
s1 move.w #lenm1,across
s2 move.b (a5)+,(a4)+
   dbf     across,s2
   dbf     down,s1

;  movea.l array1,a4
  movea.l array1,a5
  movea.l bpl,a1
   lea     byte(a1),a1
  move.l  #lenm1,down
gp5
  moveq   #mpl,across    ;length/32-1
gp4
  moveq   #31,d2         ;fill all 32 bits
gp1
  move.b  (a5)+,sum
;  move.b  sum,(a4)+
  cmpi.b  #0,sum
  beq.s   gp1
  asr.b   #2,sum
  addq.b  #1,sum
g1
   asr.b   #1,sum        ;1st color bit to x
   roxl.l  #1,d3         ; and to d3
   asr.b   #1,sum        ;2d color bit to x
   roxl.l  #1,d4         ; and to d4
   asr.b   #1,sum        ;3d color bit to x
   roxl.l  #1,d0         ; and to d0
```

MOVING?

SUBSCRIPTION PROBLEMS?

Please don't forget to let us know if you are having a problem with your subscription or if you are planning to move, please write to

Amazing Computing Subscription Questions
PiM Publications, Inc.
P.O. Box 2140
Fall River, MA 02722

Please remember, we cannot mail your magazine if we do not know where you are.

Please allow four to six weeks for processing.

This Issue On Disk:

- **ARexx Disk Cataloger**
- **GetFile Shell for True BASIC**
- **Olé—An Arcade game Programmed in AMOS**
- **Programming the Amiga in Assembly Language Part VI**
- **Porting a B+Tree Library to the Amiga**
- **Computer Simulations in Assembly Language**
- **Wrapped Up in True BASIC**

Also: Important Application Contest Information

Advertisers

* Company wishes to be contacted directly.

AC's TECH Disk
Volume 3, Number 2

A few notes before you dive into the disk!

- You need a working knowledge of the AmigaDOS CLI as most of the files on the AC's TECH disk are only accessible from the CLI.

- In order to fit as much information as possible on the AC's TECH Disk, we archived many of the files, using the freely redistributable archive utility 'lharc' (which is provided in the C: directory). lharc archive files have the filename extension .lzh.

  To unarchive a file *foo.lzh*, type *lharc x foo*
  For help with lharc, type *lharc ?*
  Also, files with 'lock' icons can be unarchived from the WorkBench by double-clicking the icon, and supplying a path.

Be Sure to Make a Backup!

CAUTION!

Due to the technical and experimental nature of some of the programs on the AC's TECH Disk, we advise the reader to use caution, especially when using experimental programs that initiate low-level disk access. The entire liability of the quality and performance of the software on the AC's TECH Disk is assumed by the purchaser. PIM Publications, Inc., their distributors, or their retailers will not be liable for any direct, indirect, or consequential damages resulting from the use or misuse of the software on the AC's TECH Disk. (This agreement may not apply in all geographical areas.)

Although many of the individual files and directories on the AC's TECH Disk are freely redistributable, the AC's TECH Disk itself and the collection of individual files and directories on the AC's TECH Disk are copyright ©1990, 1991, 1992, 1993 by PIM Publications, Inc., and may not be duplicated in any way. The purchaser however, is encouraged to make an archive/backup copy of the AC's TECH Disk.

Also, be extremely careful when working with hardware projects. Check your work, twice, to avoid any damage that can happen. Also, be aware that using these projects may void the warranties of your computer equipment. PIM Publications, or any of it's agents, is not responsible for any damages incurred while attempting any project.

```
  asr.b   #1,d0m          ;4th color bit to x
  roxl.l  #1,d1           ; and to d1
  dbf     d0,gp4          ;do 32 times
  move.l  d1,$5000(a1)    ;all 4th color bits
  move.l  d0,$1000(a1)    ;all 3d color bits
  move.l  d4,$1f40(a1)    ;all 2d color bits
  move.l  d3,(a1)+        ;all 1st color bits
  dbf     across,gp4
  lea     byteoff(a1),a1
  dbf     down,gp5
loop
  btst    #6,$bfe001      ;pressed LMB?
  bne     showit          ;branch if not

  syslib  permit
close_mem:
  movea.l array2,a1
  move.l  #size,d0
  syslib  freemem
close_cur:
  movea.l array1,a1
  move.l  #size,d0
  syslib  freemem
close_window:
  movea.l window,a0
  intlib  closewindow
close_screen:
  movea.l screen,a0
  intlib  closescreen
close_libs:
  movea.l gfxbase,a1
  syslib  closelibrary
close_int:
  movea.l intbase,a1
  syslib  closelibrary
done:
  move.l  stack,sp
  rts

  evenpc

stack   dc.l 0
gfxbase dc.l 0
intbase dc.l 0
screen  dc.l 0
window  dc.l 0
rp      dc.l 0
vp      dc.l 0
array1  dc.l 0
array2  dc.l 0
bp1     dc.l 0
bp2     dc.l 0
bp3     dc.l 0
bp4     dc.l 0
bp5     dc.l 0
k1      dc.b 0
  evenpc
k2      dc.b 0
  evenpc
gg      dc.b 0
  evenpc
diff    dc.b 0
  evenpc

graphics dc.b 'graphics.library',0
  evenpc
intuition dc.b 'intuition.library',0
  evenpc

myscreen
  dc.w 0,0,320,200,depth
  dc.b 0,1
  dc.w 0
  dc.w customscreen
  dc.l 0,0,0,0
  evenpc
mywindow
  dc.w 0,0,320,200
  dc.b 0,1
  dc.l mousebuttons
  dc.l borderless!activate!...
  dc.l 0,0
  dc.l 0
  dc.l 0,0
  dc.w 0,0,0,0
  dc.w customscreen
  evenpc
color_table:
  dc.w $000,$121,$24c,$100
  dc.w $480,$4ab,$8c4,$7a9
  dc.w $8e8,$9c7,$aa1,$b05
  dc.w $c84,$d44,$e22,$f01
  end
```

✓

Please Write to:
Bill Nee
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

Getfile
A Shell for True BASIC

by Will Steinsiek

One of the more powerful features found in True BASIC is its ability to redesign itself through the use of external Modules and Libraries. These are functions and subroutines which can either be preloaded and made part of the language itself or simply requested and called upon by any program that needs them. External Libraries, written in True BASIC, can be compiled or used as is. Compiled versions of routines written in other languages, such as C or Assembly, can also be set up for use. Once created, these subroutines can then be executed with a single line command.

True BASIC itself comes with many such Libraries. Information on how to make use of these Libraries within your own programs is found on the disk, either in a special file or in the form of comments attached to the code.

The AmigaLib is one such Library of routines. It is found in the TBLibrary folder and includes routines for speech, cycling colors on screen, and issuing commands through the CLI, and a routine for selecting a filename from a disk directory.

Unfortunately, this last routine has a problem that is referred to in the documentation. It will not work if you change screen modes with anything other than SET MODE "graphics". If, for instance, you reset the screen with SET MODE "LACEHIGH16", then calling on this routine will cause your system to crash. The problem can be resolved, however, by simply issuing the command SET MODE "graphics" before calling upon this routine to do its job.

There is an even better way, though. Instead of hoping that you remember to include the set mode command in your programming, you can write an External Library. External Libraries can, in fact, call other External Libraries in true structured programming fashion. The GetFileLib8 itself is actually a True Basic subroutine calling another machine language subroutine to do the job. Therefore we can easily design a routine that will take care of resetting the graphics mode for us before we access the filename function of the AmigaLib. While we are at it, we can also save our previous screen mode and screen, so that we can restore it as soon as we are finished.

The program GetFileLib does all that and spruces up the display a bit as well by providing some instructions for the user. It makes use of other routines found in a second Library included with True BASIC called *IFF*, which was designed to enable the user to display and use IFF graphic files such as those produced by *DeluxePaint*. As you will see, making use of this structured programming approach results in a much smaller program.

We will write GetFileLib as an External Library containing one subroutine, and sharing only one variable with the main program. All other variables within GetFileLib will remain isolated from the main program, so that any variable in our program sharing the same name used by the main program will not be accidently altered.

Creating an External Library is actually no different from the process of creating any other True BASIC program, except that it must begin with the word EXTERNAL on a line by itself at the top of the code. Convention also suggests the author include some comment lines at the beginning to identify the program, what it does, and how to use it from your main program. No other special effort is required here.

Type in Listing 1. You will note that it opens other libraries for its own use and calls routines from these in much the same way that our main program will call this routine.

When you are done, save this program as GetFileLib in the UsrLib drawer on your True BASIC program disk. Keeping your own Library routines in this particular drawer is suggested by the authors of True BASIC, but not actually required. Doing so right now, however, will make it easier for us to find it later.

The second listing will provide a test for our new External Library. It also illustrates how a very powerful program can be constructed with just a few lines using such Libraries. Save this program as IFFViewer.

Routines written in Assembly or C can also be used by True BASIC in much the same way. Once you have assembled and linked such a program, all that is missing is the proper file header, describing the program and specifying any parameters needed. A program called Finaltouch that comes with True BASIC is used to add this information to your program. It is located in the Assembly drawer on the True Basic disk.

There is more information on using your own Assembly routines within True BASIC in the Assembly.Doc file on disk 3 of your True BASIC disks, and in the Assembly drawer on the first disk there is also a sample program called MyOrd.asm. It illustrates how to tell your machine language routine where a particular string variable is located, and how to return a value to True BASIC.

Before calling such a subroutine True BASIC sets up a table in memory containing pointers for the arguments being passed to your subroutine. The start of this argument location table is found in A6, one of the address registers. Each pointer is four bytes long, and can be found by subtracting four from the value of A6. If the routine is a function, an additional pointer after all the others will point to the requested return variable.

As you can see, our main program serves as little more than a shell for calling the necessary external subroutines. The result is a very small, very capable little program.

When you are ready, run the program from within True BASIC. It will spend some time compiling itself and then present you with a scrolling list of files on the current disk. By clicking on the line labeled Path Name you can change disk drives, disks, etc. Insert a disk containing at least one graphic file. You can then scroll through the list of files and locate an IFF picture you wish to view. Double clicking on that file name will load and display the picture file. Pressing the mouse button again will return you to the file requester once more. From the file directory Select Cancel to return again to the BASIC editor.

As you can see, our main program serves as little more than a shell for calling the necessary external subroutines. The result is a very small, very capable little program.

Now that you've tried it out, feel free to modify either program. Maybe you would like to change the color scheme in the GetFileLib, or expand IFFViewer to include a slideshow option. Certainly you may want to compile the final programs. Once compiled they are essentially machine language routines with a header to tell True BASIC how to use them.

In the case of a string variable, this pointer tells the location of yet another pointer which points to the start of the string variable. In the case of a numeric variable or number, the pointer points to the location of the numeric value itself.

The following program in assembly code reads the value of a numeric variable passed to it by True BASIC and then returns that value to True BASIC. Although it does nothing very useful, it illustrates the procedure and can form the basis for inserting your own routine.

Sample Assembly Language Routine

```
move.l   (a6),a1       ;pick up ptr to numeric
    move.l   -(a6),a2      ;and ptr to output arg
    moveq    #-1,d0        ;integer -1 value (with exp = -1)
    clr.w    d0            ;clear integer part
    move.l   (a1),d0       ;get numeric argument
      {Insert your own routine here - put result in d0}
done:
    move.l   d0,(a2)       ;save in output variable
```

```
    rts                 and done
end
```

Save this code as RETURN.ASM. Using the A68K assembler you would then assemble it as RETURN.O. Using BLINK RETURN.O will complete the assembly process.

Then load True BASIC and the program called Finaltouch found in the assembly drawer. To the first question reply, DEF RETURN(N) Then give the name of your assembled program, which should be RETURN. Finaltouch will do the rest. Make sure that the final program, RETURN, is placed in the USR/Lib drawer on your True BASIC disk.

When it is ready, the following True Basic program will let you try it out.

Simple Test Program for
library "Usr/lib/return"

```
declare def return
    let t = 5
let n = return(t)
print n
end
```

While not very useful, this routine and the one called MYORD.ASM found in the assembly drawer do illustrate procedures that can be used to combine True BASIC programs with machine language subroutines. Once you know how to do this, much more interesting things are possible.

Even though True BASIC on the Amiga has a lot of features to recommend it already, there are still many things missing that would make it a more suitable Amiga programing tool. Unlike AMOS, for instance, True Basic has no facility for handling sprites. The graphic capabilities of the latest Amiga computers are unknown to it, and a host of other sound and graphic capabilities inherent within the soul of the Amiga remain just out of reach within True Basic.

True BASIC is a language that is capable of changing, however, to meet the needs of its users. As more Amiga programmers provide new building blocks that can enable it to do new tasks or to accomplish even more with less work, True BASIC will grow more accustomed to its new home on the Amiga. Thanks to its modular design, it lies within our power to uniquely tailor this flexible language to fulfill the ongoing promise of the wonderful computer platform that is waiting at our fingertips.

Listing 1

```
! Shell for Getfile$ in amigalib
!
! PURPOSE: Corrects bug in Getfile$ function by
! using a SET MODE "graphics" command before
! calling Getfile$. Saves previous mode and screen
! and restores them before returning. NOTE - Makes
! special use of MODULE IFF_library included with
! True BASIC.
!
!
! AUTHOR: Bill Steinbiss - 12/28/92
!
!
 SUB Getfile$ (filename$)
    LIBRARY "AmigaTools:IFF"          ! Open
Required Libraries
    LIBRARY "(TBLibrary)AmigaLib"
    DECLARE FUNCTION getfile$         ! Func-
tions used from Libraries
     DECLARE FUNCTION Ask_If_BMB$
     DECLARE FUNCTION Ask_If_HAM$
   ASK WINDOW lc,rt,lr,up             ! Save
Current Screen
    BOX KEEP lc,rt,lr,up in screen$
    ASK CURSOR row,col
    ASK MODE oldmode$
    LET BMB$ = Ask_If_BMB$
   LET HAM$ = Ask_If_HAM$
    CALL Save_SYS_colors              ! Routine
found in IFF Library
    ! Reset screen mode to Workbench (uses
workbench colors)
    !       And create shell adding instructions
for user
    SET MODE "graphics"
!    PLOT .05,0;.95,0;.95,.9;.05,.9;.05,0
    SET CURSOR 2,28
    PRINT "DIRECTORY OF FILE NAMES"
    SET CURSOR 4,17
    PRINT "To Change Disks Click on PATH NAME,
Delete Field"
    SET CURSOR 5,17
    PRINT "   And Enter Drive (DF0: DF1: RAM:
etc.)"
    SET CURSOR 6,17
    PRINT "To Select File Double Click on File
Name"
```

Statement of Ownership, Management and Circulation 1A. Title of Publication: AC's Tech for the Commodore Amiga. 1B. Publication No.: 10537929. 2. Date of Filing: 10/1/92. 3. Frequency of Issue: Quarterly. 3A. No. of Issues Published Annually: 4. 3B. Annual Subscription Price: $44.95 US. 4. Complete Mailing Address of Known Office of Publication: P.O. Box 2140, Fall River, MA 02722-2140. 5. Complete Mailing Address of the Headquarters of General Business Offices of the Publisher: P.O. Box 2140, Fall River, MA 02722-2140. 6. Full Names and Complete Mailing Address of Publisher, Editor and Managing Editor: Publisher, Joyce A. Hicks P.O. Box 2140 Fall River, MA 02722; Editor, Donald D. Hicks P.O. Box 2140 Fall River, MA 02722; Managing Editor, Donald D. Hicks P.O. Box 2140 Fall River, MA 02722. 7. Owner: PiM Publications, Inc. P.O. Box 2140 Fall River, MA 02722; Joyce A. Hicks P.O. Box 2140 Fall River, MA 02722. 8. Known Bondholders: None. 9. For Completion by Nonprofit Organizations Authorized to Mail at Special Rates: Not Applicable. 10. Extent and Nature of Circulation: (X) Average No. Copies Each Issue During Preceding 12 Months, (Y) Actual No. Copies of Single Issue Published Nearest to Filing Date. 10A. Total No. Copies. (X) 7,691 (Y) 7,560. 10B. Paid and/or Requested Circulation: 1. Sales through dealers nad carriers, street vendors and counter sales (X) 3,216 (Y) 4,086. 2. Mail Subscription (X) 1,525 (Y) 1,539. 10C. Total Paid and/or Requested Circulation: (X) 4,741 (Y) 5,625. 10D. Free Distribution by Mail, Carrier or other Means Samples, Complimentary, and other Free Copies: (X) 44 (Y) 60. 10E. Total Distribution: (X) 4,785 (Y) 5,685. 10F. Copies Not Distributed: 1. Office Use, Left over, Unaccounted, Spoiled after Printing (X) 1,520 (Y) 1,860. 2. Return from News Agents. (X) 1,386 (Y) 15. 10G. Total: (X) 7,691 (Y) 7,560.

```
    SET CURSOR 7,[illegible]
    PRINT "   Or Click Once and Select"
    SET CURSOR 11,8
    PRINT "PATH NAME:"
    SET CURSOR 14,8
    PRINT "FILE NAME:"
    LET filename$ = getfile$(150,70,t5,"SELECT")
    SET mode oldmode$                          ! Restore
previous screen
    IF EHBS = "YES" then CALL EHB_ON  ! Routine
found in IFF Library
    IF HAMS = "YES" then CALL HAM_ON  ! Routine
found in IFF Library
    CALL Restore_SYS_Colors                ! Routine
found in IFF Library
    SET window [illegible]
    BOX SHOW screen$ at [illegible]
    SET CURSOR row,col
END SUB
```

Listing 2

```
    IFF Viewer Program using External Libraries to
select file for viewing and to display IFF Image
 Selection Cancel or any non IFF file will return to
BASIC
! AUTHOR: Will Steinsiek  -  12/28/92
LIBRARY "[illegible]"          ! Libraries used,
including [illegible]
LIBRARY "[illegible]"          !    Called by
GetFile.lib
   CALL GetFile(filename$)        ! Gets file name
into shared variable
   CALL [illegible]
  CALL Read_IFF_image(filename$,"IFF","IFF",0,1)
   DO until r = 0                 ! Wait for mouse
button to clear
      GET MOUSE a,b,c
   LOOP
   DO until r <> 0                ! Wait for mouse
button to be pressed
      GET MOUSE a,b,c
   LOOP
LOOP
END

GetFile.lib for True BASIC
Will Steinsiek - [illegible]
212 Cardenas
Albuquerque, NM 87108
Phone: (505) [illegible]
```

✓

Please Write to:
Will Steinsiek
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

ARexx Disk Cataloger

ARexx is an interpreted interprocess communication (IPC) programming language. That's a mouthful. For novice Amiga users, IPC sounds difficult and probably not worth their time to learn and use. If I had said ARexx is an easy-to-learn programming language which can automate AmigaDOS chores accomplished on a regular basis, then ARexx might interest the novice programmer.

This article is about an ARexx disk Cataloger program that manipulates AmigaDOS to produce a text file containing information about the floppy disks (or hard drives) that you want cataloged. This program was designed for novice ARexx programmers and is limited to AmigaDOS commands. The best way to learn ARexx programming is to study program examples and use those examples to write your own ARexx programs.

The ARexx programming language, part of the Amiga Operating System (OS) 2.0 or purchased separately for OS V1.3, is for the masses. If you know anything about BASIC programming, then learning ARexx is reasonably easy. If you are new to any type of programming, you may need to purchase a book about ARexx programming. I highly recommend *Using ARexx on the Amiga* by Chris Zamara and Nick Sullivan (published by Abacus). This book is ideal for the novice and experienced ARexx user.

One day, I needed a listing of the contents of 11 disks. Since I misplaced the disk catalog program that I sometime used, I had to use the AmigaDOS command line interface (CLI) and my directory utility (SID) to catalog the disks. I was constantly using the keyboard, the arrow keys, and the mouse to execute the following CLI and SID commands:

- DIR > RAM:Holdit df1: opt a <CR>, which redirects directory listing to the RAM file, 'Holdit'
- AE <CR>, which loads my text editor
- Load 'RAM:Holdit' text file into text editor and preform the following;
    a. insert blank lines at the top and bottom
    b. insert the disk volume name
    c. insert the remaining free disk space
- JOIN RAM:Holdit and Catalog as Catalog.new (this makes a new file, 'Catalog.new' from the files 'RAM:Holdit' and 'Catalog'. Catalog is the accumulation of the all the floppy disk that I want cataloged.)
- Use SID (a directory utility) to;
    — DELETE RAM:Holdit and RAM:Catalog
    — RENAME RAM:Catalog.new to RAM:Catalog

Then I would start the process over for another disk. While I was creating my catalog file, I thought that a small ARexx program could accomplish this process. At the time, I knew very little about the ARexx language, but its commands appeared uncomplicated. Now to discuss various ARexx programming techniques I discovered while writing the ARexx Cataloger program.

An AmigaDOS script could do what the ARexx Cataloger does, but it would be much more difficult for an AmigaDOS script to handle the Cataloger's flexible response to user input and its ability to determine the peripheral resources available on a given Amiga computer system.

Listing 1 is the Cataloger program. The line numbers in the program are for reference purposes only and are not part of the ARexx Cataloger program. When you type in the program, omit the line numbers. The code is documented to explain program flow and ARexx commands.

The ARexx Cataloger evolved as I learned about ARexx. The final program has nine sections. The sections are;

1. Mandatory comment statement at the beginning of the file. It is unnumbered.
2. Create a customized input/output window for the program, lines 1 to 13.
3. Setup, which has three parts;
    a. Initialize constants, lines 14 to 25,
    b. Check system for resources and DOS commands, lines 26 to 100, and
    c. Load the system disk devices, lines 102 to 119.
4. Get the source disk drive, lines 120 to 147.
5. Get the destination device and filename, lines 148 to 235.

6. Read the disk in the source drive and build the Catalog, lines 236 to 327.

7. Print the results, lines 3286 to 380.

8. Clean up and close out, lines 381 to 386.

9. Program procedures, 387 to 449.

As you study the sections, you will notice all the sections support section six, which is the code that creates the catalog file. In order for section six to produce a catalog file, the preceding sections must initialize variables and accept keyboard inputs, so it can adapt to any Amiga computer system setup.

The first section of the program is the required ARexx program comment line (denoted by the start comment symbol /* and the end comment symbol */). Lines 4, 6, and 74 are examples of comment usage within a program. The only required comment is the very first line of an ARexx program. When ARexx encounters the /* symbol it ignores everything until it encounters the */ symbol. Other comments throughout the program are to document program flow. I recommend liberal use of comments to document program flow, even if you create ARexx programs for your use only. Following the comment is section two of the Cataloger program, the customized input/output window.

The customized window allows you to create a window sized to your ARexx program requirements. STDIN and STDOUT are filenames assigned by ARexx to the CLI window it opens when you pass a program to ARexx. All interactive input/output (I/O) is handled by the STDIN and STDOUT files. Since the ARexx Cataloger requires constant user interaction, its customized window is the same size and attributes of the normal 640 x 200 hi-res Workbench screen (reference line 3). ARexx treats this customized window as a file and redirects the input and output to the window. A modification of the customized window code can result in two different windows with STDOUT assigned to one window and STDIN assigned to the other window. Listing 2 is an example of this type of modification.

Section three of the Cataloger program initializes the variables used in the program and determines the peripheral resources (disk drive devices).

A one dimensional array _cmdr and three escape sequence variables are initialized at the beginning. The _cmdr array is a list of AmigaDOS command names. The program uses the PATH, INFO, DELETE, JOIN, and ECHO AmigaDOS commands. The Cataloger is flexible in that it searches all the system loaded paths for the commands. It does this by loading a text file in RAM (line 40) and then uses the information in the file to create a second one dimensional array called path. Listing 3 is an example of the text file created by Cataloger from which it determines the search path for the AmigaDOS commands. Following the _cmdr array are the escape sequence variables color_1 through color_3.

Cataloger uses escape sequences (see page 7-45 of the Amiga OS 2.04 manual for a listing of Standard Escape Sequences for Console Window) to change the ARexx window pen color in the middle of a string. This was necessary when a string contained two or more different colors. The 1bx part of the escape sequence (line 23 to 25) is the hexadecimal representative of the escape key, which is where the term escape sequences originates. An interesting problem surfaced when I used the color_1 through color_3 variables with the center and say functions.

Clockwise from the top: 1. Catalog destination requestor. 2. Read-disk information screen. 3. Catalog source requestor. 4. Next disk requestor.

The center function, as used in the Cataloger, centers a string based on the width you pass to the function. For example:

```
var = center("Center this string",80)
```

will return a string to var with 31 leading spaces, followed by "Center this string", and then 31 more spaces. If the 80, which is the width, is changed to 60, then "Center this string" would have 21 leading and trailing spaces. When center is combined with the say statement, the string is centered and printed on the STDOUT (assuming you used the appropriate width for the STDOUT window). Due to the window borders, center functions in the ARexx Cataloger are based on 77 characters width.

When escape sequences are imbedded in a string their length must be included in the width or the string will not be centered on the screen. Reference line 17 of Listing 1. The escape sequence 16x[32m is five non-printable characters in length. The 16x counts as one character. To ensure the text string in line 17 centers on the STDOUT, then the five characters must be added to the required 77. This is why 82 was used in the center function of line 17. For every escape sequence, add five to the 77 width and the strings will remained centered in the STDOUT window. If I used 83 for the width in this example (one more than what I can get on the screen), then ARexx would follow the text string with a blank line. This is an simple method to get a line of text on the screen followed by a blank line with one ARexx statement.

The || symbols are used throughout the program to put together (concatenate) text strings and escape sequences. Reference line 126 for an extreme example of || symbol usage.

Pen color procedures are at the end of the Cataloger program (lines 405 to 415; named color1, color2, and color3). The color procedures use escape sequences with the AmigaDOS ECHO command to change the pen color which uses the *E symbol (rather than the 16x used by ARexx) to represent the escape key. They are used to change the pen color for a whole line of text.

The program generates another one-dimensional array during the initialization process called drive. in section three. It uses the AmigaDOS INFO command (line 103) to create a text file in RAM. Listing 4 is a sample of the INFO command generated text file. The program searches through the file one line at a time until it locates the line that has 'Unit' as the first word (line 109). This line is discarded, then it loads the drive array with the system drive unit(s) (line 115). The program exits the load drive loop when it locates the blank line before 'Volumes available' (line 112). Cataloger uses the drive array to determine if the source drive and destination filename are valid devices and filename for your system.

If you use a system utility, which allows the computer system to read both Amiga and MS-DOS disks on the same drive, the Cataloger program will read and catalog both types of disks. There is a problem associated with this dual drive identity that is error trapped (line 207 to 230) and will cause the Cataloger to display a rather lengthy discussion of the problem. For example, if the Amiga DF1: disk drive is also known to the computer system as MS-DOS MD1: disk drive (i.e., two logical devices sharing the same physical drive), and DF1: is entered as the source drive and MD1:Catalog is entered as the destination, then the program will error trap the problem and explain it on the screen. If you avoid this dual drive conflict, the ARexx Cataloger will even catalog your MS-DOS disks without incident, if you have a MS-DOS utility installed.

The Cataloger will catalog as many disks as your storage space will allow. Press CONTROL D to exit the read disk routine. It could take up to 2.5 seconds to exit after depressing CONTROL D, so be patient. When you exit the read disk routine, you are asked if you want to print the cataloged file. If you answered yes (i.e., upper or lower case 'Y'), set your printer and press the return key to start the printing. The program creates a header for the catalog file which is a summation

of the disk read by the program. Listing 5 is an example of the Cataloger printout.

You can modify the print portion of the Cataloger and make it a stand-alone ARexx program which will print out a previously cataloged text file. This is a simple process. Load the Cataloger into your text editor or word processor and delete the first line comment of the program and lines 15 through 21? lines 26 to 350, and line 379. This should leave the customized window, the escape sequences, the print routine, and the procedures portion of the original program. You can also delete the color1 and fatal_error procedures, since they are not use in the print routine. When this is complete, add the following to the first 13 lines of your new ARexx program and save it as CatalogPrint.rexx. If you used a word processor, ensure you save it as a text file.

/* Catalog printout - usage; rx CatalogPrint <filename> */
parse arg dest /* assigns <filename> to dest */
if length(dest) = 0 | ~exists(dest) then do /* <filename> doesn't exist */? call screen_cls? call skip_lines(6)? call color3? say center('Usage: rx CatalogPrint <filename>',77)? say? say center('Where <filename> is pathname to Catalog file',77)? delay(200)? call screen_cls? exit end

The finale to the Cataloger project is to make the program selectable from the Workbench environment. This is easy to do, if you have a directory utility like the shareware SID program. Workbench can also duplicate the icon by selecting Icon Copy from the Workbench menu. Once the Shell icon is duplicated, rename it Cataloger. Then highlight the icon by clicking on it once and then simultaneously press the right Amiga key and the I key (upper or lower case) for Icon Information. This brings up the Information Requester. Erase any information in the Default Tool window and enter rx as the default tool. Select and delete any information in the tool types window. Save the this information and relocate the icon to any directory in your system. When you double click the Cataloger icon, ARexx will look in the Rexx directory for the program and execute it.

ARexx is a programming language for the masses. Even a novice ARexx programmer can write impressive programs. The exciting aspect of ARexx is its flexibility. Individual creative ideas are the only limit for what ARexx can do for you. Your creative ideas and viewpoint could modify the ARexx Cataloger program and make it better than it is or you could write other cosmic programs. Once you jump into ARexx, and realize that it can manipulate applications (like *excellence!, SuperBase Pro 4, Proper Grammar*, etc.) as easy as it manipulates AmigaDOS then the possibilities are endless. Try ARexx, you'll like it.

Listing 1

```
/*Catalog.rexx script - usage: rx Catalog.

  Required material: [illegible]
   AREXX programming language
   rexxsupport.library
   AmigaDOS commands
      info
      delete
      join
      dir
      echo
*/
1 close('STDIN')  /* close current standard input */
2 close('STDOUT') /* close current standard output */
3 if open('STDOUT','con:0/0/640/200/ARexx Disk
Cataloger','W') then do  /* open a new window */
4   PRAGMA('*','STDOUT')  /* redirect to new window */
5   if ~open('STDIN','*','R') then do
6     /* if open & reassign not successful, reopen old
STDIN & STDOUT */
7      close('STDOUT')
8      PRAGMA('*')  /* open '*', console handler */
9     open('STDOUT','*','W')
10     open('STDIN','*','R')
11      exit  /* exit out of the script */
12      end
13 end  /* of 'if open('STDOUT', ... */
14 /* initialize */
15 call screen_cls
16 call skip_lines(8)
17 say center('1b'x'[32m' || 'Initializing Program',92)
18 _cmd.1 = 'INFO'
19 _cmd.2 = 'DELETE'
20 _cmd.3 = 'JOIN'
21 _cmd.4 = 'DIR'
22 /* escape sequences */
23 color_1 = '1b'x'[31m'
24 color_2 = '1b'x'[32m'
25 color_3 = '1b'x'[33m'
26 /* is AREXX support library available and loaded? */
27 if ~exists('libs:rexxsupport.library') then do
28   call screen_cls
29   do i=1 to 8    /* skip eight lines */
30       say ' '
31    end
32  errtext = 'Cannot find libs:rexxsupport.library!'
33   call fatal_error
34   exit      /* exit from script */
35 end
36 /* load support library */
37 address command 'rx [illegible] rexxsupport.library 0 -30 0'
38 /* load search path into RAM: file */
39 address command 'path > ram:holdit'
40 open('file','RAM:holdit','R') /* open file for reading
*/
41 count = 0  /* determine number of paths to search */
42 do while ~EOF('file')  /* while path data available */
43   /* load path into path array element */
```

```
path.count = strip(readln('file'))
  if right(path.count,1) ~= ":" then do
    path.count = path.count"/"
  end
  count = count + 1 /* increment path element counter */
end
close('file') /* close the temporary file in RAM: */
success = 0   /* initialize flag */
do n = 1 to 3 /* search for ECHO, INFO, DELETE, and LIST */
  flag = -1   /* set flag */
  do i=1 to count
    /* if found in pathway, reset flag */
    if exists(path.i || cmd.n) then flag = 1
  end
  if flag = -1 then do
    /* if flag not reset, then command not present */
    success = -1 /* command not found flag set */
    call screen_cls
    call skip_lines(8)
    say center('1b'x'[1m' 'You are missing the command' '1b'x'[1m' cmd.n || '1b'x'[0m',92)
    delay(250) /* wait five seconds, then continue */
    call screen_cls
  end
end
if success = -1 then do
  /* one of the commands was not found */
  errtext = "One or more command programs missing from current search path(s)."
  call fatal_error
  exit /* exit script */
end
/* If OS 2.0 is used, then ECHO and PATH are resident to the system. Otherwise, check for path to echo and assume that the command PATH is available from the system.
   Determine if OS 2.0 is being used.
*/
a = showlist('L','exec.library',,'a') /* lib address */
b = offset(a,20)  /* add the offset */
c = import(b,2)   /* get two characters from address */
d = c2d(c)        /* convert to decimal number */
if d < 37 then do /* version 1.3 found below 37 */
  count = 1       /* check for ECHO in version 1.3 */
  do forever
    if exists(path.count || 'echo') then do
      leave     /* exit the loop */
    end
    if count = count then do
      /* ECHO not found in any system search paths */
      call screen_cls
      call skip_lines(5)
      say center(color_3 || 'E R R O R',83)
      say center(color_1 || 'I can not find the ' || '1b'x'[1m' 'ECHO' || '1b'x'[0m command',83)
      say center(color_2 || 'Press RETURN key to quit',83)
      parse pull dummy /* wait for RETURN key */
      call screen_cls
      exit /* exit script */
    end
    count = count + 1 /* inc element counter */
  end
end

/* Determine drive devices on the host system. Redirect INFO data to ram:. Then open as read only, with pointer at the top of file */
address command 'info >RAM:Rdidir'
call open('file','RAM:Rdidir','R')
flag = 0 /* initialize program flow flag */
max_element = 0 /* initialize device counter */
do while ~EOF('file')
  line = strip(upper(Readln('file')))
  if left(line,4) = "UNIT" then flag = 1 /* change program flow within the loop */
    if flag = 1 then do
      /* look for blank line in the file just prior to mounted devices listing */
      if length(line) = 0 then leave /* when blank line located */
      max_element = max_element + 1  /* increment element counter */
      /* load drive designator */
      drive.max_element = strip(upper(word(line,1)))
    end
end
close('file')
address command 'delete >NIL: RAM:Rdidir' /* delete file RAM:Rdidir */
/* Get the source drive */
call screen_cls
do forever /* get source disk drive */
  flag = 0 /* initialize exit flag */
  call screen_cls
  call skip_lines(6)
  say center(color_1 || 'Enter source drive, like, ' || color_3 || 'RAM:' || color_1 || ', ' || color_3 || 'DF1:' || color_1 || ', ' || color_3 || 'DH0:' || color_1 || ', etc.',113)
  Options Prompt '' || color_2 || left(' ',18) || "Enter disk drive to catalog => " || color_3
  parse pull CatDrive /* get user input for source drive */
  call screen_cls
  CatDrive=strip(UPPER(CatDrive)) /* upper case */
  /* see if user put ':' on the end of device name */
  x=pos(':',CatDrive,1)
  if x=0 then CatDrive = CatDrive':' /* put the ':' on the selection, if user doesn't */
  do n=1 to max_element
    if drive.n = CatDrive then do
      flag = 1 /* match found, valid device */
      leave    /* exit loop */
    end
  end
  if flag = 1 then leave /* the forever do loop */
  call skip_lines(6)
  say center(color_3 || CatDrive || color_2 || " is not a drive currently located on your system.",88)
  delay(150) /* wait three seconds */
  say center(color_1 || "Try again.",83)
  delay(75)
  call screen_cls
end
/* Get destination file for the catalog */
call screen_cls
```

```
call skip_lines(4)
say center(color_2 || 'It is best to locate your catalog file in ' || color_1 || 'RAM:' || color_2 || '.',83)
say center('especially if you have only one disk drive.',78)
delay(150)
call screen_cls
do forever  /* get a valid destination file */
  call skip_lines(8)
  say center(color_2 || 'Enter full pathname for your catalog file.',81)
  say center(color_1 || 'Example: ' || color_3 || 'RAM:Catalog ' || color_1 || 'or ' || color_3 || 'DH0:Catalog' || color_1 || ', etc.',103)
  options prompt color_2 || left(' ',15) || 'Enter catalog destination file -> ' || color_3
  parse pull dest  /* get destination from STDIN */
  call screen_cls
  dest = strip(upper(dest))
  j = pos(':',dest,1)  /* find position of colon */
  select
     when length(dest) = j then do /* colon is last */
         /* indicates only a device was entered */
         call skip_lines(8)
        say center('You must select a file in the destination pathname!',77)
        say center(color_1 || 'Try again!',83)
         delay(200)
         call screen_cls
         end  /* 'when length(dest)' selection */
     when j = 0 then do /* colon not present; error */
         /* indicates no device was entered */
         call skip_lines(8)
         say center(color_1 || dest || color_2 || ' is an...',88)
        say center(color_2 || 'Incorrect drive/file selection!',83)
        say center(color_1 || 'Try again!',82)
         delay(200)
         call screen_cls
         end  /* of 'when j = 0' selection */
     when j > 1 then do /* format correct; see if a valid filename was used */
         /* assumes a device & file were entered */
         flag = -1  /* initialize flag */
         do n=1 to max_element  /* check device name */
            if drive.n = left(dest,j) then flag = 1 /* device is on system */
         end
         if flag = -1 then do
           /* pathname device, not part of system */
           call skip_lines(6)
           say center(color_1 || left(dest,j) || color_2 || ' is an',88)
           say center(color_2 || 'Invalid drive selection!',83)
           say center(color_1 || 'Try again!',82)
            delay(200)
            call screen_cls
            end
         else if abbrev(dest,CatDrive) then do
            /* CatDrive string is in dest string */
            call skip_lines(8)
            say center(color_2 || 'Sorry, but you can not use your ' || color_1 || 'catalog drive ' || color_2 || 'as your ' || color_1 || 'destination',97)
            delay(200)
            call screen_cls
            end
         else if ~abbrev(dest,CatDrive) then
            /* valid system destination selected, check for logical and physical disk drive conflict */
            PRAGMA('W','N')  /* turn off disk requestors */
            if open('Cat',dest,'W') then do /* open filenamed 'dest' */
                call screen_cls
                call skip_lines(2)
                call color2
                say left(' ',15) || 'Source Drive' || left(' ',15) || 'Destination Drive'
                call color1
                say left(' ',21) || CatDrive || left(' ',27-length(CatDrive)) || dest
                call color1
               say center('Invalid device used, most likely due to logical device',77)
               say center('and physical device conflicts. This could occur when using a',77)
              say center('MultiDOS type utility programs. These allow you to read both',77)
               say center('AmigaDOS and MS-DOS disks in the same physical Amiga drive.',77)
                call color2
               say center('Ensure you are not using a MS-DOS logical device',77)
               say center('as part of your destination filename.',77)
                call color1
               Options Prompt center('Press RETURN key to continue',77)
                parse pull dummy
                call screen_cls
                end
              else do
                close('Cat')
                leave /* exit forever loop */
            end  /* of 'if open('Cat',dest ....' */
       end  /* of 'when j > 1' & 'if statements */
     otherwise
         /* Nope, don't exit */
  end /* end of select */
end /* of forever loop */
/* Create the Catalog file */
PRAGMA('W','N')  /* turn off disk requestors */
open('Cat',dest,'W')  /* open filenamed 'dest' */
Writeln('Cat',' ')  /* blank line separator */
close('Cat')
open('test','RAM:Lines','W')
/* make a file with three blank lines */
Writeln('test',' ')
Writeln('test',' ')
Writeln('test',' ')
Close('test')
open('head','RAM:Header','W')
/* Create Header file */
Writeln('head',' ')
Writeln('head',left(' ',5) || 'Catalog file prepared on 'Date()'.')
Writeln('head',' ')
```

```
Writeln('head',left(' ',8) || 'DISK #  Volume Name')
Writeln('head',left(' ',8) || left('-',6,'-') || left('-',27,'-'))
Writeln('head',' ')
close('head')
/* log disk until CONTROL D is pressed */
DISK = 0 /* initialize disk counter */
call screen_cls
signal on BREAK_D /* CONTROL D interupt on */
old_dir = "SYS:" /* initialize to SYS: device */
vol_name = "SYS:"
do forever /* until CONTROL D is selected */
   if exists(CatDrive) then do /* is disk in drive */
      call screen_cls
      call skip_lines(6)
      address command 'info > RAM:info_hold' CatDrive
      open('file','RAM:info_hold','R')
       do x=1 to 4  /* 4th line contains 'CatDrive' volume information */
          data = Readln('file')
       end
      close('file')
      n = words(data)  /* how many words in "data"? */
      vol_name = strip(word(data,n)) || ':'  /* last word in string is volume name */
     if vol_name = 'present:' then do /* occurs when disk is not present in CatDrive */
         vol_name = old_dir  /* escape for no disk in 'dest' */
       end
      if vol_name ~= old_dir then do  /* wait for the old disk to be removed */
        diskBL_used = value(word(data,3)) /* blocks */
        diskBL_free = value(word(data,4)) /* blocks */
        disk_size = strip(word(data,2))  /* K or M */
        sz = right(disk_size,1) /* get 'K' or 'M' */
       size = substr(disk_size,1,length(disk_size)-1) /* numeric size of disk */
          if sz = "M" then do
             size = size * 1000 /* MegaByte Size Disk */
          end
        disk_free = strip(value(size * (diskBL_free/(diskBL_used + diskBL_free)))) /* convert to string */
        n = pos('.',disk_free) /* decimal point */
          if n ~= 0 then do
           disk_free = substr(disk_free,1,n-1) || 'K' /* get only KiloBytes */
          end  /* end of 'if n ~= 0 ......*/
        old_dir = vol_name  /* to ensure it will not take back to back disk of the same name */
          DISK = DISK + 1   /* inc disk counter */
        open('head','RAM:DiskHeader','A') /* append info to header file */
          d = center(' ' || DISK || ' ',8) /* take the center eight characters */
          a = '  ' || d || '  ' || vol_name
        Writeln('head',a)
        close('head')  /* close header file */
       say center(color_2 || 'The disk in' CatDrive 'is Volume ' || color_3 || vol_name,88)
        say center(color_2 || ' Label it as disk number ' || color_3 || DISK,88)
        say center(color_1 || "Getting information from disk",83)
        open('Cat',dest,'A') /* open, append header */
        Writeln('Cat','DISK Number' DISK || ',  Volume Name -->' vol_name)
        Writeln('Cat','Free disk space' disk_free 'KB')
        Writeln('Cat',' ')
        Close('Cat')
/*
  using AmigaDOS, create a file in RAM: called '_dir_file', then join the three files in RAM: (the catalog file, _dir_file, and Lines as the file RexxCat, which is the complete catalog file to date. Then delete the old catalog file 'dest'. Once completed, copy RAM:RexxCat to 'dest' as the new, up to date, catalog file. Clean up by deleting RAM:RexxCat and RAM:_dir_file. Leave 'RAM:Lines' alone.
*/
        address command 'dir > RAM:_dir_file' CatDrive 'opt a'
        address command 'join > NIL: ' dest 'RAM:_dir_file RAM:Lines as RAM:RexxCat'
        address command 'join > NIL: ' dest 'RAM:_dir_file RAM:Lines as RAM:RexxCat'
        address command 'copy > NIL: RAM:RexxCat To' dest 'quiet'
        call screen_cls
      end  /* end of 'if vol_name ~= ......*/
   end  /* end of 'if exists(CatDrive) ...... */
   call screen_cls
   call skip_lines(8)
   call color1
   /* second method to center up text on STDOUT */
   say left(' ',20) || 'Please, place next disk in device ' || color_3 || CatDrive
   call color1
   say left(' ',37) || 'or'
   say ''
   say center('Press ' || color_2 || 'CONTROL D ' || color_1 || 'to ' || color_2 || 'EXIT',92)
   delay(100)   /* wait 2 seconds for multitasking */
   call screen_cls
end   /* end of forever loop */
BREAK_D: /* forever loop exit point */
signal off BREAK_D /* turn off error trapping */
address command 'join >NIL: RAM:DiskHeader ' dest 'as RAM:RexxCat'
address command 'copy > NIL: RAM:RexxCat To' dest 'quiet'
address command 'delete > NIL: RAM:DiskHeader'
address command 'delete > NIL: RAM:Lines RAM:RexxCat'
address command 'delete > NIL: RAM:info_hold RAM:_dir_file'
if DISK == 0 then do /* ensures no exit without reading at least one disk */
   address command 'delete > NIL:' dest  /* erase empty file that was created, but nothing was put in it */
   call Control_Exit
   exit  /* allow no printing */
end
do forever  /* until the Y, y, N, or n selected */
   call screen_cls
   call skip_lines(8)
```

```
  options prompt color_2 || left(" ",11) || 'Do you
want to print catalog file just created (Y/N)? '||
color_1
  parse pull decision
  call screen_cls
  decision = strip(upper(decision))
    if decision = "Y" | decision = "N" then leave
end /* end of do forever */
call screen_cls
if decision = "Y" then do
  do forever /* ensure printer online */
     p = open('printer','PRT:','W')
      /* open printer just like any other device */
       select
         when p == 0 then do  /* PRT: not ready */
             call screen_cls
             call skip_lines(6)
            say center(color_2 || 'Please place the
printer on line so I can print the file.',80))
             say center(color_2 || "I'll try again in
five seconds!",79)
             delay(250)      /* multitask wait for 5
seconds */
          end
          otherwise
             leave     /* printer is ONLINE */
       end  /* of 'select' */
  end /* end of do forever */
  linecount = 0 /* initialize PRT: line counter */
  page = 0      /* initialize PRT: page counter */
  open('Cat',dest,'R') /* open catalog file */
  call prt_header
  do while ~EOF('Cat') /* keep going until EOF */
     line = Readln('Cat')
     linecount = linecount + 1 /* inc line counter
*/
     if linecount >= 60 then do /* allow 1" margins
*/
        Writeln('printer','0C'x) /* send formfeed */
        call prt_header
         end
     else /* print line with six leading spaces */
        Writeln('printer',left(' ',6) || line)
  end  /* end of 'if' and 'do while' statements */
end  /* while ~EOF() */
/* cleaning up all items */
if exists('printer') then do
  close('Cat')  /* if not closed, close now */
  Writeln('printer','0C'x) /* send formfeed */
  close('printer') /* close printer */
end
call Control_Exit
close('STDIN') /* close custom input window */
close('STDOUT')  /* close custom output window */
PRAGMA('*')   /* open "*", console handler */
open("STDOUT","*","W") /* Redirect output to ARexx
*/
open("STDIN","*","R")  /* Redirect input to ARexx */
exit  /* end of program */
/* procedures */
prt_header: procedure expose linecount page
    page = page + 1      /* inc page counter */
  Writeln('printer',' ')
  Writeln('printer',' ')
  Writeln('printer',left(' ',6) || 'Catalog File,
page -' page)
  Writeln('printer',' ')
  Writeln('printer',' ')
   linecount = 5
return
/* following procedures use escape sequences to
change the pen color */
color1: procedure /* default text color 1 */
   address command 'echo "*E[31m"'
return

color2: procedure /* default text color 2 */
   address command 'echo "*E[32m"'
return

color3: procedure /* default text color 3 */
   address command 'echo "*E[33m"'
return

/* escape sequence to clear screen */
screen_cls: procedure
   address command 'echo "*E[1;1H*E[J"'
return
/* skip some "lines" on the screen */
skip_lines: procedure
   parse arg lines  /* lines passed as an argument */
   do ext to lines
        say   /* blank line */
   end
return
/* error message for script termination */
fatal_error: procedure expose errtext
   call screen_cls
   call skip_lines(6)
   call color2
   say center('F A T A L   E R R O R',77)
   call color3
   say center(errtext,77)
   call color1
   say center('Press RETURN key to exit ',77)
   parse pull dummy
   call screen_cls
return
Control_Exit: procedure
   call screen_cls
   call skip_lines(6)
   call color1
   say left(' ',28) || 'Exiting ARexx Catalog Script'
   delay(100)  /* two seconds then exit */
   call screen_cls
return
/* end of Catalogue.rexx script */
```

Listing 2

```
/* WindowSmall.rexx - Customized window demo */
close('STDIN')
close('STDOUT')

if open("STDOUT","con:1/1/640/100/Window 1/","W") then do
  PRAGMA("*","STDOUT")
  if ~open("STDIN","con:1/100/640/100/Window 2","R") then
do
```

```
    close('STDOUT')
     PRAGMA('***')
    open('STDOUT','***','W')
    open('STDIN','***','R')
     exit
     end
end
say center('Testing STDOUT Window #1',77)
say center('Enter 'Testing' in Window 2 to exit',77)
options prompt '                    '
parse pull variable  /* get input from STDIN */
say
say center(variable,77)
close('STDIN')
close('STDOUT')
PRAGMA('***')  /* open '*' the console handler */
open('STDOUT','***','W')  /* reopen the ARexx window */
open('STDIN','***','R')
```

Listing 3

```
Current_directory /* first line of PATH output */
RamDisk:
BootDrive:C
BootDrive:c/c_ak2
BootDrive:Utilities
BootDrive:Rexx
BootDrive:System
BootDrive:S
BootDrive:Prefs
BootDrive:WBStartup
BootDrive:PC
BootDrive:Tools
BootDrive:Tools/Commodities
BootDrive:
C>              /* last line */
```

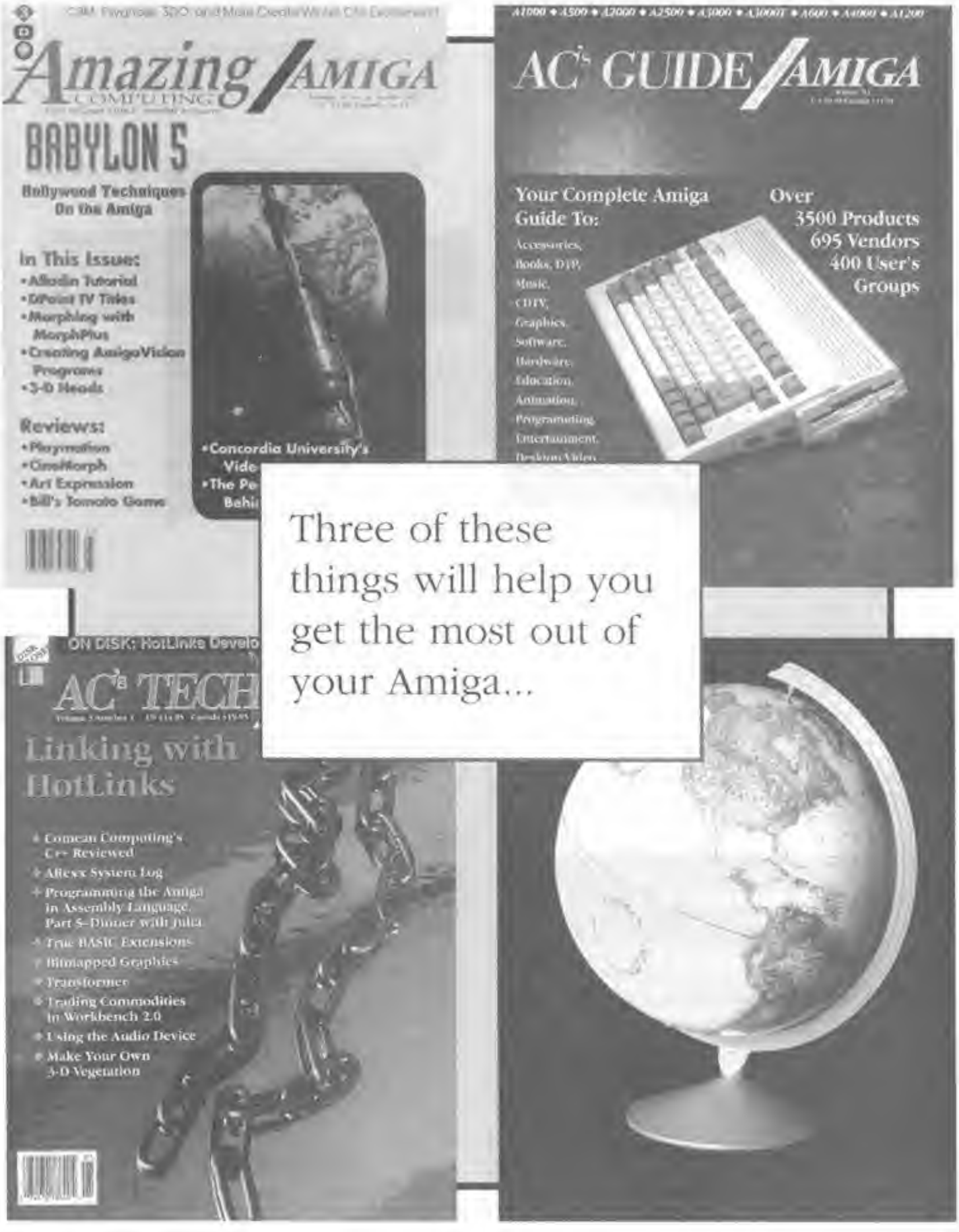
Listing 4

```
/* first line is blank for LNM:command */Mounted disks:
Unit    Size   Used   Free Full Errs   Status   Name
MD1:    No disk present
MD0:    No disk present
RAM:    51K     51      0  100%    0  Read/Write RamDisk
DH0: 89088K  21521   32795  41%    0  Read/Write BootDrive
DF0:    No disk present
DF1:    No disk present
DH1:   28M   94030    5837  91%    0  Read/Write FastDrive
DH2:   14M   14311    2003  88%    0  Read/Write Work
Volumes available:
RamDisk [Mounted]
Work [Mounted]
FastDrive [Mounted]
BootDrive [Mounted]
/* last line is blank */
```

Listing 5

```
    Catalog File prepared on 26 Jan 1991.

    DISK #   Volume Name
    ------   -----------

       1     Quarterback_Reports:
       2     ARexx_Article_HD:

DISK Number 1, Volume Name -> Quarterback_Reports:
Free disk space 516K Kb

    DIR_Listing (dir)
      Dir_091291.lzh
  .info                          Disk.info
  Quarterback                    Quarterback.info
  Report_DH0_1.lzh               Report_DH0_2.lzh
  Report_DH0_3.lzh               Report_DH0_4.lzh
  Report_DH0_5.lzh               Report_DH0_6.lzh
  Report_DH0_7.lzh               Report_DH0_8.lzh
  Report_DH1_1.lzh               Report_DH1_2.lzh
  Report_DH1_3.lzh               Report_DH1_4.lzh
  Report_DH1_5.lzh               Report_DH2_1.lzh
  Report_DH2_2.lzh               Report_DH2_3.lzh
 Report_DH2_4.lzh


DISK Number 2, Volume Name -> ARexx_Article_HD:
Free disk space 705K Kb

 2WindowDemo.rexx
AMAZ_Article_ARexx_1.dor
 AMAZ_Article_ARexx_1.txt       Cataloger.info
 Cataloger.rexx                 CatalogPrint.rexx
 Listing_1                      Listing_2
 Listing_3                      Listing_4
 Listing_5                      Pipeman.info

LISTING 5.
```

Please write to:
T. Darrel Westbrook
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722


Amazing AMIGA COMPUTING
BABYLON 5
Hollywood Techniques On the Amiga
In This Issue:
•Aladdin Tutorial
•DPaint IV Titles
•Morphing with MorphPlus
•Creating AmigaVision Programs
•3-D Heads
Reviews:
•Playmation
•CineMorph
•Art Expression
•Bill's Tomato Game
•Concordia University's
•The Pe
Behi
A1000 • A500 • A2000 • A2500 • A3000 • A3000T • A600 • A4000 • A1200
AC's GUIDE AMIGA
Your Complete Amiga Guide To:
Accessories, Books, DTP, Music, CDTV, Graphics, Software, Hardware, Education, Animation, Programming, Entertainment,
Over 3500 Products 695 Vendors 400 User's Groups
Three of these things will help you get the most out of your Amiga...
ON DISK: HotLinks Develo
AC's TECH
Linking with HotLinks
Comeau Computing's C++ Reviewed
Aliens System Log
Programming the Amiga in Assembly Language, Part 5–Dinner with Julia
True BASIC Extensions
Bitmapped Graphics
Transformer
Trading Commodities in Workbench 2.0
Using the Audio Device
Make Your Own 3-D Vegetation

—**Assembly** *continued from page 23*

```
  moves.w #256,maxcount
   bra      no_message
do_set512
  moves.w #512,maxcount
   bra      no_message
do_set1024
  moves.w #1024,maxcount
   bra      no_message
zoom
   andi.l   #$0000ffff,d5          ;make mouseX a word
   andi.l   #$0000ffff,d6          ; and mouseY
   move.l   d5,startx              ;save them
   move.l   d6,starty
   move.l   startx,d0
   mode     complement             ;XOR color
lmb_down
   cIm      lmdl
;  cmpi.l   #mousebuttons,d3
;   bne.s   lmb_down
   cmpi.w   #selectup,d3            ;lmb must be up
   beq      lmb_up
lmdl
   andi.l   #$0000ffff,d5
   andi.l   #$0000ffff,d6
   move.l   d5,endx                ;save ending coordinates
   move.l   d6,endy
   box     startx,starty,endx,endy,5
   delay    1                      ;optional
   box     startx,starty,endx,endy,5
   bra      lmb_down
lmb_up
   mode     jam1                   ;restore draw mode
   imul     xscale,startx
   add.l    xc,d0
   move.l   d0,newxc               ;new xc

   imul     xscale,endx
   add.l    xc,d0
   sub.l    newxc,d0
   flitdp
   movedp   d0,d6
   flitdp   320
   movedp   d0,d2
   movedp   d6,d0
   divdp
   fixdp                           ;new xscale
   tst.l    d0
   bne.s    new_xscale
   beep
   moveq    #1,d0                  ;make it at least 1
new_xscale
   move.l   d0,xscale
   move.l   newxc,d0
   move.l   d0,xc

   move.l   yscale,d0
   moveq    #0,d1
   move.l   #200,d1
   sub.l    endy,d1
   move.w   d0,d2
   mulu     d1,d2
   swap     d0
   mulu     d1,d0
   swap     d0
   clr.w    d0
   add.l    d2,d0
   add.l    yc,d0
   move.l   d0,newyc

   move.l   yscale,d0
   moveq    #0,d1
   move.l   #200,d1
   sub.l    starty,d1
   move.w   d0,d2
   mulu     d1,d2
   swap     d0
   mulu     d1,d0
   swap     d0
   clr      d0
   add.l    d2,d0
   add.l    yc,d0
   sub.l    newyc,d0
   flitdp
   movedp   d0,d6
   flitdp   200
   movedp   d0,d2
   movedp   d6,d0
   divdp
   fixdp
   tst.l    d0
   bne.s    new_yscale
   beep
   moveq    #1,d0
new_yscale
   move.l   d0,yscale
   move.l   newyc,d0
   move.l   d0,yc
   bra      showit
no_message
   addq.w   #1,down                ;down one space
   cmpi.w   #200,down              ;all way down yet ?
   bne      ml2                    ;branch if not
ml1
   move.l   xloc,d1
   add.l    xscale,d1              ;increase xloc by xscale
   move.l   d1,xloc
   addq.w   #1,across              ;over one space
   cmpi.w   #320,across            ;all way across yet ?
   bne      ml1                    ;branch if not
   bra      check_for_message

close_window
   closemenu
   closewindow
close_screen
   closescreen
close_libs
   closelib dpmath
close_gfx
   closelib gfx
close_dos
   closelib dos
close_int
   closelib int
done
   move.l   stack,sp
   rts

   evenpc

stack      dc.l  0                 ;reserve storage loca-
```

```
tions
gfxbase    dc.l  0
intbase    dc.l  0
dosbase    dc.l  0
dpmathbase dc.l  0
           even
                                   ;library names
gfx       dc.b  'graphics.library',0
  evenpc
int       dc.b  'intuition.library',0
  evenpc
dos       dc.b  'dos.library',0
  evenpc
dpmath    dc.b  'mathieeedoubbas.library',0

myscreen:
   dc.w 0,0,320,200,depth        ;depth is 5
   dc.b 1,2
   dc.w 0
   dc.w customscreen
   dc.l 0,0,0,0
   evenpc
mywindow
   dc.w 0,0,320,200
   dc.b 31,18
   dc.l mousebuttons|menupick|mousemove
;IDCMP flags
   dc.l activate|smartrefresh|borderless|mousereport
;window flags
   dc.l gadget1,0
   dc.l 0
   dc.l 0,0
   dc.w 0,0,0,0
   dc.w customscreen
   evenpc
xc        dc.l  $c0000000          ;-2 * scale
newxc     dc.l  0
yc        dc.l  $c0000000
newyc     dc.l  0
xscale    dc.l  $00666666          ;(4 / 320) * scale
yscale    dc.l  $00a3d70a          ;(4 / 200) * scale
xloc      dc.l  0
yloc      dc.l  0
aloc      dc.l  0
bloc      dc.l  0
asqr      dc.l  0
asqr4     dc.l  0
bsqr      dc.l  0
bsqr4     dc.l  0
sum       dc.l  0
sum4      dc.l  0
diff      dc.l  0
diff4     dc.l  0
juliaa    dc.l  0
juliab    dc.l  0
startx    dc.l  0
starty    dc.l  0
endx      dc.l  0
endy      dc.l  0
across    dc.w  0
down      dc.w  0
sign      dc.w  0
julia     dc.w  0
  evenpc
colormap                            ;my new palette
  dc.w $000,$f0f,$d0f,$b0f,$90f,$70f,$50f,$d0f
  dc.w $10f,$00f,$03d,$05b,$079,$097,$0b5,$0d3
  dc.w $0f0,$1f0,[illegible]
  dc.w $fd0,[illegible]
   evenpc
menu0
  makemenu menu0,'Project',menu1,0,1
   makeitem
menu0item0,'Mandelbrot',menu0item1,0,[illegible],'M'
   makeitem
menu0item1,'Julia_Set',menu0item2,10,$57,$4,'J'
   makeitem
menu0item2,'Coordinates',menu0item3,20,$56,,'C'
  makeitem menu0item3,'QUIT',,30,$56,,'Q'
  makemenu menu1,'Display',,90,1
   makeitem
menu1item0,'ItCount',,0,$92,,,menu1item0subitem0
    makesubitem
menu1item0subitem0,'64',menu1item0subitem1,0,$137,$1e
    makesubitem
menu1item0subitem1,'128',menu1item0subitem2,10,[illegible]
    makesubitem
menu1item0subitem2,'256',menu1item0subitem3,20,[illegible]
    makesubitem
menu1item0subitem3,'512',menu1item0subitem4,30,[illegible]
   makesubitem menu1item0subitem4,'1024',,40,[illegible]
gadgets
  makestrgadget
gadget1,'Xleft',gadget2,40,100,100,8,30,51,50,1,15
  makestrgadget
gadget2,'Ytop',gadget3,110,75,100,9,30,51,50,2,15
  makestrgadget
gadget3,'Xright',gadget4,180,100,100,8,30,51,50,3,15
  makestrgadget
gadget4,'Ybot',gadget5,110,125,100,8,30,51,50,4,15
  makestrgadget
gadget5,'JuliaA',gadget6,40,150,100,8,30,51,50,5,15
  makestrgadget
gadget6,'JuliaB',,180,150,100,8,30,51,50,6,15
   evenpc
gadget1buffer dc.b '-2.0000000000',0  ;default values
   evenpc
gadget2buffer dc.b '2.0000000000',0
   evenpc
gadget3buffer dc.b '2.0000000000',0
   evenpc
gadget4buffer dc.b '-2.0000000000',0
   evenpc
gadget5buffer dc.b '0.0000000000',0
   evenpc
gadget6buffer dc.b '0.0000000000',0
 end
```

✓

Please Write to:
William P. Nee
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

—**OLE** *continued from page 13*

```
'
* see if hot spots are within 6. pick frontal image.
climb 50 lines slowly.
* this will remain constant regardless of cpu because of
applied vblanking.
* every second pixel of climb we will check will collision
with a bull.
* set climb flag. increment operations deck level #,
reset 'got prize' flag.
* note how we make assign bool expressions to r6 and r9,
then checking both
* in the same 'if' test. thus we save one jump. keeping
within the 3 to a
* loop limit).
'
O: Let R6=R7>6; Let R9=R7< -6; If R6|R9 J A: L A=7;
   F R0=1 T 25  L Y=Y-2: If BC(0,1,4) J P:  M PU:
   L R2=1; L RO=RO+1; L RL=0; J A;
'
* matador collided with bull. move him off screen. set
'gored' flag
'
P: F R0=1 To 15 L K=X-15; L Y=Y-15: M RJ; L R2=0; L
R6=0; L R9=0;
  L R9=0; L RO=1; J A;


' chan 1 to bob 1
' bottomost bull
L X=15; L Y=196;
A: Anim 0,(4,9)(5,15); M 290,0,RA
Anim 0,(2,9)(3,15); M -290,0,RA
Jump A;


' chan 2 to bob 2
' 2nd bottom most bull
L X=104; L Y=146;
A: Anim 0,(2,9)(3,15); M -290,0,RA
Anim 0,(4,9)(5,15); M 290,0,RA
Jump A;


' chan 3 to bob 3
' 2nd from top bull
L X=15; L Y=96;
A: Anim 0,(4,9)(5,15); M 290,0,RA
Anim 0,(2,9)(3,15); M -290,0,RA
Jump A;


' chan 4 to bob 4
' topmost bull
L X=104; L Y=46;
A: Anim 0,(2,9)(3,15); M -290,0,RA
Anim 0,(4,9)(5,15); M 290,0,RA
Jump A;


' chan 5 to bob 5
' bottom prize
' assign value from setprizes procedure to this channel's
'X'
L X=RM;
' if collision permitted, check for one, else pause, and
retest flag.
A: If R1=0 J B: P; J A;
' if collision detected, set loop flag, and test for skill
level
B: If BC(5,2,0) J C: P: J A;
C:   L R1=1;
' set appropo prize image value. incr score per current
skill level
L A=BC+19; L R0=RC+1; L RM=R0*10+RM; J A;


' chan 6 to bob 6
' second prize from bottom
' assign value from setprizes procedure to this channel's
'X'
L X=R1;
' if collision permitted, check for one, else pause, and
retest flag.
A: If RL=0 J B: P; J A;
' if collision detected, set loop flag, and test for skill
level
B: If BC(6,2,0) J C: P: J A;
C:   L RL=1;
' set appropo prize image value. incr score per current
skill level
L A=BC+19; L R0=RC+1; L RM=R0*10+RM; J A;


' chan 7 to bob 7
' 2nd from top prize
' assign value from setprizes procedure to this channel's
'X'
L X=RJ;
' if collision permitted, check for one, else pause, and
retest flag.
A: If RL=0 J B: P; J A;
' if collision detected, set loop flag, and test for skill
level
B: If BC(7,2,3) J C: P: J A;
C:   L RL=1;
' set appropo prize image value. incr score per current
skill level
L A=BC+19; L R0=RC+1; L RM=R0*10+RM; J A;


' chan 8 to bob 8
' top prize
' assign value from setprizes procedure to this channel's
'X'
L X=RK;
' if collision permitted, check for one, else pause, and
retest flag.
```

```
A: If BL=0 ; B; T; : A:
' If collision detected, set loop flag, and test for skill
level
B: If BC(9,0,8) ; C; B: ; A:
C: ; L,RL=1;
' Get appropriate image value, and score per current
skill level
LA=RC+19; LRA=RC+1 ; LRM=RM*10+RM: LAH=1 ; A:
```

Listing Two

```
'
' Ole(AMOS)by T.J. Kaholeum September 10, 1992
'                         after Pest/Amiga by John Gilmore
'   9 Skill levels  - Bulls go faster, Matador goes
slower. 8-)
'
'
LoadIff "Ole/Title.pic",2
LoadIff "Ole/Ole.pic",1
Load "Ole/OleSprites.abk" : Rem Takes bank 1.
Load "Ole/OleAmal.abk" : Rem Takes bank 4.
Load "Ole/OleSamples.abk" : Rem Takes bank 5.
Load "Ole/OleMusic.abk" : Rem Takes bank 3.
Screen 2
Music 1
Auto View Off
'
Screen Open 1,320,200,16,Lowres
Screen Copy 0 To 1 : Rem Get a background pic.
'
Get Sprite Palette : Flash Off : Hide : Double Buffer
Screen 0 : Rem Make this current
Fade 3 : Rem so we can fade it to black.
Screen 2
Synchro Off : Rem To detect collisions in AMAL strings.
'
Dim LAD(4) : Rem Horizontal coords for 3 ladders
Dim PRIZE(4) : Rem Horizontal coords for the prizes.
'
' All bobs being assigned an AMAL channel must first be
referenced.
' Bobs 1,2,3,4 are the bulls, bottom deck to top.
' Bobs 5,6,7,8 are the prizes. Bottom deck to top. See
Proc SETPRIZES
'
Bob 0,256,250,0 : Rem Matador. All 3 bobs drawn off screen
to start with.
Bob 1,16,290,2 : Bob 2,320,240,2 : Bob 3,16,290,2 : Bob
4,304,240,2
Bob 5,290,290,10 : Bob 6,360,250,10 : Bob 7,370,290,10 :
Bob 8,290,290,10
'
'Bobs 9,10,11 are the ladders, bottom deck to top. See
Proc SETLADDERS
'Bobs 12,13,14,15 are the remaining Matador icons. See
Proc SETICONS
'
Channel 0 To Bob 0 : Channel 1 To Bob 1 : Channel 2 To Bob
2
Channel 3 To Bob 3 : Channel 4 To Bob 4 : Channel 5 To Bob
5
Channel 6 To Bob 6 : Channel 7 To Bob 7 : Channel 8 To Bob
8
'
' RA=BULL SPEED. RC=Skill level (0-8). RD=Operations
deck (0-3).
' RE, RF, RG are ladder X's.  RH, RI, RJ, RK are prize
X's.
' RL=Got prize flag. RM=Score. RN=MATADOR SPEED.
RO=Gored flag.
' RP=4th Level done flag  RQ=Jumping delay time.
RR=Score increase.
'
'                 Image Nos. for prizes at each of the 9
levels.
'
'  BAG=10 : BALLOON=11 : SHADES=12 : CONE=13 : CANNON=14
'  BEER=15 : LOGO=16 : HOTDOG=17 : WHISKEY=18
'
'                 Sound File Sample Numerical Equivalents
'
BOING=1 : CROWD=2 : GLASS=3 : HIGH=4 : HORN=5 : LAUGH=6 :
OH=7 : PRIZE=8 : YELL=9
'
' Here follows the "main()" program.
' These instructions are given once and done when game
begins.
'
Ink 3,5
Text 2,9,"Ole! Amos 1.0"
Amal 0,0 : Amal 6,5 : Amal 5,6 : Amal 7,7 : Amal 8,8 : Rem
Like Amal 8,88
Amal On : Rem Bull is under program control so as to always
start at edges.
Proc SETICONS : Rem Little faces in the title bar.
Volume 63
Amreg(Asc("D")-65)=0 : Rem Reset current ladder X.
Amreg(Asc("L")-65)=0 : Rem Reset 'got prize' flag
Amreg(Asc("M")-65)=0 : Rem Reset scoreboard
Amreg(Asc("O")-65)=0 : Rem Reset 'gored' flag.
Amreg(Asc("R")-65)=5 : Rem Reset 'scored' flag.
Text 270,9,"SCORE 0"
A=5 : Rem User gets 5 lives to lose.
'
' Anim channels can be assigned only to currently-existing
bobs.  Draw
' the bulls, prizes and man 'off screen' to avoid prema-
ture displays.
' To be sure our bulls always start from the edges of the
display, we turn
' their channels off, restarting them each 'gore' or new
level.  This means
' reexecution from the first line of code where X is set
at the display edges.
' The matador is handled similarly to help avoid 'colli-
sions' on resets.
'
Wait 1000
Music Off
Auto View On : Rem Blackened screen 0 is front and 'vis-
ible'.
For B=0 To 8 : Rem 9 skill levels.
   Amal Off 0 : Amal Off 1 : Amal Off 2 : Amal Off 3 :
Amal Off 4
   Screen To Back : Rem Put current screen (1) behind.
```

```
Bob screen 2 front.
  Screen 4 : Rem Make screen 4 current.
  Ink 4,2 : Rem print blue on tan bg.
  Text 90,120,"Press Fire Button When Ready"
  Fade 1 To -2 : Rem Flash screen 4 with color.
  Wait 30 : Rem Give it a chance to fade.
  Sam Play 3,DE
  Do
     Wait Vbl
     If Fire(1) Then Exit
  Loop
  Fade 4 : Wait 60 : Rem Fade screen 4 to black.
  Auto View Off : Rem Turn off display while we draw and
fill.
  Screen 1 : Rem Make hidden screen current. We draw our
face on it.
  Fade 4 To -1 : Rem Flash hidden Screen 1 with sprite
colors.
  Wait 15 : Rem Must allow flash time, or color transi-
tions unreliable.
  Ink 1,0 : Rem Transparent text by printing on background
  Text 52,175,"                    "
  Ink 4,0 : Rem back to blue on white for scoring in the
hidden bar.
  Amal 0,0 : Amal 1,1 : Amal 2,2 : Amal 3,3 : Amal 4,4 :
Rem Must fire amal programs
  Amal On 0 : Amal On 1 : Amal On 2 : Amal On 3 : Amal On
4
  Amreg(Asc("C")-65)=B : Rem inform AMAL of the number of
current skill level
'
  SPEED=(100-(B*8)) : Rem Determined empirically by
author & Tom Reed? ...
  Amreg(Asc("A")-65)=SPEED : Rem Bull from 100-40 in
increments of 8.
'
  If SPEED<70
    Amreg(Asc("Q")-65)=(SPEED/10) : Rem Jumps to passed
...speed...
  Else Amreg(Asc("Q")-65)=1 : Rem But passing ... at
higher speeds.
  End If
'
  MANSPEED=(11-B)/2
  If MANSPEED<2
    MANSPEED=1
  End If
  Amreg(Asc("M")-65)=MANSPEED : Rem Man speed 5-1 from
6 5 4 4 3 3 2 2
'
  Proc SETPADDERS : Rem position for random distribution
of bars
  Proc SETFENCES[8+(B)]
  Proc SETMATADOR
  Screen To Front : Rem Bring current screen (1) front.
  Screen 0 : Rem Quickly bleach screen 0 for future fade
in's.
  Home
  Screen 1 : Rem screen 1 current again.
  Auto View On : Rem Let's see it!
  Do
    Synchro : Rem Gives each of our AMAL channels a
'shot'
    Wait Vbl
    '
    If Amreg(Asc("O")-65)=1 : Rem We've been 'gored'
      Sam Play 3F,HOING
      Wait 50
      Bob Off 9+10 : Rem Erase an icon
      Dec A : Dec B : Rem Lose one life. Skill level
restored on loop.
      Amreg(Asc("O")-65)=0 : Rem Reset 'gored' flag.
      Sam Play 3F,YELL
      If A=1 : Rem No more lives
        Exit
      End If
      Fade 16
      Wait 100
      Sam Play 3F,OLABS
      Wait 150
      Exit
    End If
    '
    If Amreg(Asc("R")-65)>Amreg(Asc("H")-65)
      Sam Play 3F,PRICK
      Text 220,8,"SCORE"+Str$(Amreg(Asc("R")-65))
      Amreg(Asc("H")-65)=Amreg(Asc("H")-65)
    End If
    '
    If Amreg(Asc("F")-65)=1 : Rem Ready for next skill
level.
      Amreg(Asc("F")-65)=0 : Rem Reset skill level flag,
and reloop.
      Sam Play 15,CORN
      Sam Play 15,CHORD
      Wait 50
      Fade 10
      Wait 150
      Exit
    End If
'
  Loop
'
  If A=1 : Rem Out of lives (score)
    B=1 : Rem Cause B to ... the brands.
  End If
```

```
'
  If B=8
    Sam Play $F,HIGH
    Wait 50
    Sam Play $9,HERE
    Sam Play $a,CROWD
    Fade 15
    Wait 325
    End
  End If
Next B
Sam Play $A,LANDS
Wait 50
Sam Play $F,LAND
Fade 15
Wait 325
'
End
'
Procedure SETLADDERS
  Shared LAD()
  B=195
  For A=0 To 4
   '                                    Ladders lows: X = 12-308,
Y = 98, 146 & 196.
    LAD(A)=Rnd(288)+12
    Amreg(A+4)=LAD(A) : Rem Assign ladder X's to R8, R9,
R0.
    Bob A+8,LAD(A),B-(A*50),1 : Rem Trick allows ascend-
ing bottom upward
  Next A
  Amreg(Asc("D")-65)=0 : Rem Reset channel D ladder climb
coordinate
End Proc
Procedure SETPRIZES[LEVEL]
  Shared PRIZE()
  B=190
  For A=0 To 3
    PRIZE(A)=Rnd(296)+8
    Amreg(A+7)=PRIZE(A)
   Bob A+1,PRIZE(A),B-(A*50),LEVEL
  Next A
  Amreg(Asc("L")-65)=0 : Rem Reset 'get prize' flag
End Proc
Procedure SETTURNS
  For A=0 To 3
    Bob A+12,(A*14)+120,2,5
  Next A
End Proc
Procedure SETMATADOR
  Bob 0,167,146,6 : Rem For completeness. Repeats the
Bob consistently.
End Proc
```

✓

Please write to:
Thomas J. Eshelman
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722

—BTree *continued from page 31*

```
showrec.o: showrec.c $(HEADERS)
    $(CC) $(FLAGS)    showrec.c

upindex.o: upindex.c $(HEADERS)
    $(CC) $(FLAGS)    upindex.c

mkkey.o: mkkey.c $(HEADERS)
    $(CC) $(FLAGS)    mkkey.c

opendb.o: opendb.c $(HEADERS)
    $(CC) $(FLAGS)    opendb.c

closedb.o: closedb.c $(HEADERS)
    $(CC) $(FLAGS)    closedb.c

filename.o: filename.c $(HEADERS)
    $(CC) $(FLAGS)    filename.c

chgextnt.o: chgextnt.c $(HEADERS)
    $(CC) $(FLAGS)    chgextnt.c

flushdb.o: flushdb.c $(HEADERS)
    $(CC) $(FLAGS)    flushdb.c

# Removed getdisk.o and getcurdr.o for
AMIGA conversion
# getdisk.o: getdisk.c $(HEADERS)
#     $(CC) $(FLAGS)    getdisk.c

# getcurdr.o: getcurdr.c $(HEADERS)
```

✓

Please write to:
John Bushkara
c/o
AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

AC's TECH Back Issue Index

AC's TECH Volume 1 Number 1

Adapting Mattel's Power Glove to the Amiga by Paul King and Mike Cargal
AmigaDOS for Programmers by Bruno Costa
AmigaDOS, EDIT and Recrusive Programming Techniques by Mark Pardue
An introduction to Interprocess Communication with ARexx by Dan Sugalski
An Introduction to the ilbm.library by Jim Fiore
Building the VidCell 256 Grayscale Digitizer by Todd Elliott
Creating a Database in C Using dBC III by Robert Broughton
FastBoot: A Super BootBlock by Dan Babcock
Magic Macros with ReSource by Jeff Lavin
Silent Binary Rhapsodies by Robert Tiess
Using Intuition's Proportional Gadgets from FORTRAN 77 by Joseph R Pasek

AC's TECH Volume 1 Number 2

A Mega and a Half on a Budget by Bob Blick
Accessing Amiga Intuition Gadgets from a FORTRAN Program Part II-Using Boolean Gagdets by Joseph R Pasek
Adding Help to Applications Easily by Philip S Kasten
CAD Application Design: Part I - World and View Transforms by Forest W Arnold
Interfacing Assembly Language Applications to ARexx by Jeff Glatt
Intuition and Graphics in ARexx Scripts by Jeff Glatt
Programming the Amiga's GUI in C Part I by Paul Castonguay
ToolBox Part I: An Introduction to 3D Programming by Patrick Quaid
UNIX and the Amiga by Mike Hubbart

AC's TECH Volume 1 Number 3

Accessing the Math Co-Processor from BASIC by R P Haviland
C Macros for ARexx by David Blackwell
CAD Application Design Part II by Forest W Arnold
Configuration Tips for SAS-C by Paul Castonguay
Hash for the Masses: An Introduction to Hash Tables by Peter Dill
Programming for HAM-E by Ben Williams
Programming the Amiga's GUI in C-Part II by Paul Catonguay
The Development of an AmigaDOS 2.0 Command Line Utility by Bruno Costa
Using RawDoFmt in Assembly by Jeff Lavin
VBRMon: Assembly Language Monitor by Dan Babcock
WildStar: Discovering An AmigaDOS 2.0 Hidden Feature by Bruno Costa

AC's TECH Volume 1 Number 4

GPIO Low Cost Sequence Control by Ken Hall
Language Extensions. Strings of Type StringS by Jimmy Hammonds
Programming the Amiga's GUI in C Part III by Paul Castonguay
Programming with the ARexxDB Records Manager by Benton Jackson
State of Amiga Development Denver DevCon Address by Jeff Scherb
STOX An ARexx Based System for Maintaining Stock Prices by Jack Fox
The Development of a Ray Tracer Part 1 by Bruno Costa
The Varafire Solution Build Your Own Variable Rapid Fire Joystick by Lee Brewer
Using Interrupts for Animating Pointers by Jeff Lavin

AC's TECH Back Issue Index

AC's TECH Volume 2 Number 1

AudioIProbe-Experimets in Synthesized Sound with Modula 2 by Jim Olinger
CAD Application Design Part III by Forest W Arnold
Implementing an ARexx Interface in Your C Program by David Blackwell
Low-Level Disk Access in Assembly by Dan Babcock
Programming a Ray Tracer in C Part II by Bruno Costa
Programming the Amiga in 680x0 Assembler Part I by William P Nee
Programming the Amiga's GUI in C Part IV by Paul Castonguay
Spartan-Build Your Own SCSI Interface for Your Amiga 5000/1000 by Paul Harker
The Amiga and the MIDI Hardware Specification by James Cook
Writing Protocols for MusicX by Daniel Barrett

AC's TECH Volume 2 Number 2

Amiga Voice Recognition by Richard Horne
Animated Busy Pointer by Jerry Trantow
Blit Your Lines by Thomas Eshelman
Copper Programming by Bob D'Asto
Dynamically Allocated Arrays by Charles Rankin
Implementing an ARexx Interface in Your C/Program Part 2 by David Blackwell
Iterated Function Systems for Amiga Computer Graphics by Laura Morrison
Keyboard I/O from Amiga Windows by John Baez
MenuScript by David Ossorio
Programming the Amiga in Assembly Language Part 2 by William P Nee

AC's TECH Volume 2 Number 3

Backup_DOC by Werther Pirani
CAD Application Design Part 4 by Forest W Arnold
HighSpeed Pascal by David Czaya
PCX Graphics by Gary L Fait
Programming the Amiga in Assembly Language Part 3 by William P Nee
Programming the Amiga's GUI in C Part 5 by Paul Castonguay
Understanding the Console Device by David Blackwell

AC's TECH Volume 2 Number 4

Advanced Scripting by Douglas Thain
Entropy in Coding Theory by Joseph Graf
Fast Plots by Michael Griebling
Getconfirm() by John Baez
In Search of the Lost Windows by Phil Burke
No Mousing Around by Jeff Dickson
Programming the Amiga in Assembly Language part 5 by William P Nee
Putting the Input Device to Work by David Blackwell
Quarterback 5.0 a Review by Merrill Callaway
Tape Drives by Paull Gittings
The Joy of Sets by Jim Olinger
True BASIC Extensions by Paul Castonguay

```
      next a
   else
      plot text, at X + LM, Y : String$
               ! USING ASPECT OR HIRES & INTERLACE?
      ! THE DOUBLE PRINTING TO KILL FLICKER
      if Yoff <> 0 then
         plot text, at  X + LM, Y + YOFF : String$
      end if
   end if
end sub
```

You do not just show an image. You call a sub whose job it is to show images at a place you desire. This sub may have its own independent work to do. Let the image subs worry about the image details. Do not burden the text subs with image overhead. Do not mix jobs. First plot the image in the desired place by passing the buck to an image routine.

If you map the space available to text, line by line in terms of the total space minus that taken up by an image, then you have in these two main subs just about everything you need for word wrap around images.

Now all that is needed is a sub to place an image within a requested location, determine the left and right boundaries available to text as we go from top to bottom, and to call the parse routine to get that amount of text that will fit each line and then the justify routine to plot it according to the rules we picked, line by line, top to bottom.

At the end of each stopping point (bottom of page or form feed etc.) call for an image update from the image routine. That is, always pause in image mode. If that image plotting routine can detect mouse and key presses (if so enabled) then it can report those it does not understand back to the caller and act on those it does understand.

Our image engine knows what to do with arrow key presses as well as a variety of 'play' command key presses. If that image happens to hold 16 frames, then we can play it or step through it at will and still return text scrolling key presses back to the caller sub.

Oh yeah. The image plotting sub should take notes as to what image it was on and what was pressed and what point the mouse was pointing at. Maybe the caller sub can use that information. Can't hurt. It would be wise to give the caller sub the ability to disable or enable some of the image handling features. Pass a key.

Problem: The picture image nearly always dictates the color palette. How can we possibly figure, in advance, with hundreds of pictures to load, that text will show up well against the backdrop color and that it won't go nuts in a HAM environment even when we label details within the HAM image?

Let's look a one solution but first consider two.
We need to place an image, then define from top to bottom how many lines there are and the left and right bounds of each line. Two main ways exist. First, just start at the top and figure each line as you go looking at the bottom margin to be sure not to go to far. The included sub works this way. It sets an upper zone left and right and a lower zone left and right. When the image is placed, those values get set. We could use an alternate method of setting up an array to hold the left and right limits and the vertical position of the line. This latter method is fast, but carries the array overhead which isn't bad. The array method would allow columns of text and central placement of images. The text printing would merely stay within the array limit set for that page. The latter was not used for no good reason. It was determined.

```
   Def SwapChars$(S$,CH1$,CH2$)
      If CH1$ <> CH2$ then
         let T$ = ""
         Do while pos(S$,CH1$) > 0
            let T$ = T$ & Before$(S$,CH1$) & CH2$
            let S$ = After$(S$,CH1$)
         loop
         let SwapChar$ = T$ & S$
      else
         let SwapChar$ = S$
      end if
   end def
end sub

! 'NoTabs' In Progress
! Strip tabs out of 'C' code and -> 3 spaces each.
! The usual 'C' editors use tabs alot. I prefer spaces.
! I can do better code conversions with all spaces.
         EXTERNAL
         SUB NoTab_DO(line$(), arg$)
            FOR I = 1 to Ubound(line$)
      LET p = pos(line$(i),CHR$(9))
       IF p > 0 then
          DO while p > 0
             LET line$(i)[p:p] = "   "
             LET p = pos(line$(i),CHR$(9))
          LOOP
       END IF
    NEXT I
         END SUB

! 'Ask_dir'
! ABSTRACT:  Reports the name of the current directory
! in the command window. Will echo if ECHO is activated.
! The compiled version is 'AskDir' is in the TBDo drawer.
         ! SYNTAX:    DO AskDir
         EXTERNAL
         sub Ask_Dir(line$(),arg$) ! arg$ is ignored.
   Library "(Toolkit)higbdos*"
   Library "(AmigaTools)dos*"
   Library "(AmigaTools)amiga*"
            call askdir(dirname$)
   cause error 1, "Directory is " & dirname$
end sub
```

Notice that a 'cause error' is used to break out of the sub and creates an err string. Err strings are automatically reported in the command window. That's cheap. I know. But, this is a very easy way to get short informational text to the command window.

```
! 'Change_dir'
! ABSTRACT: Changes & reports the name
!           of the current directory.
! Saved as compiled code 'ChangeDir' in the TBDo-directory.
! SYNTAX:    DO ChangeDir, "New Directory Name"
           EXTERNAL
           sub Change_Directory(line$(),NewName$)
   Library "(Toolkit)bigdos*"
   Library "(AmigaTools)dos*"
   Library "(AmigaTools)amiga*"
               if NewName$ = "" then
      cause error 1, "Use: Do CD, pathname"
   end if
               call askdir(OldName$)
               when error in
      call Chdir(NewName$)
   use
      cause error 1, Extext$ & " Dir=" & OldName$
   end when
               cause error 1, "Directory is " & NewName$
end sub
```

Here, the same strategies are used. An error trap 'When error in' is used to trap an error to allow formation of a meaningful message. 'Extext$' is a True BASIC string that can be checked at any time. It contains "" or text relating to a trapped error by True BASIC. Here we put the error text out, as is, but with additional information concatenated (&).

```
! "Shell$to_Do"
! The compiled version is saved as 'Shell' in the
```

however to use a method which would be most suited to HAM images and avoiding fringing without repair work, and to handle XSpees images which demand very careful vertical tracking and need text vertical offset double printing to eliminate eye destroying flicker (worse than interlace alone).

Here is what we do. Place the image in any of four corner positions. Plot a frame around it if the image carries such a flag or if requested by the caller (This frame is the simplest way to kill fringing in HAM). Set colors to that requested by the caller but allow the caller to defer to color choices encoded in the image itself. Invisibly, the image has encoded information which tells all subs what text and background color combinations work best, whether the image is 3-D, what frame color is best, and other optional goodies.

After placing the image, start the top and lay down one line of text at a time, hyphenating and justifying according to flags set. At text end or window bottom stop and replot the image in an image control sub. That sub waits for mouse click or key press. If the input is image control it does what is asked. If it is unfamiliar, it returns to the text wrapping sub. If the text wrapping sub does not recognize the key press it returns to the image control sub. Round and round we go if strange key presses are given. However, if the key press indicates forward or backward text scroll, then tracking pointers to the text are used to replot the window text and pause again in the image control sub.

The text wrap sub does more than blindly print the parsed text. It reads the text for key words and control sequences. These words can bring up a text input requester with a message such as a question. The text input from the user is stored in an answer text array to be passed back to the main program on conclusion of the sub activity. That array also carries data indicating where the mouse was pointing and what frame of the image was shown at the time of the answer.

Because each frame can be a unique image, and because each frame can encode data about itself it is easy to show a series of ANIMal images and ask "Point to a duck's bill." If the duck is image #5 and the x,y image pixel coordinates are within the set tolerance of the embedded coordinates (or color) for "Duck bill", then the program knows that the answer was correct. The program need not even know about ducks, their bills, anything specific, just that the point and click matched the query.

In normal use you do not call this sub directly, but call others with easier names which in turn call this one, setting most of the flags and values for you. Either way, here is a breakdown of the many arguments to this sub before we dissect the sub itself:

```
    sub WrapTextAndQueryANIM(&rush$(),&&.TextPtr,TextPtr,
TextNum,BackNum, WindowFrame,BrushFrameNum,Quad,
HyphPct,JustPct,Margins,LineSpacing,
&.&,&,&,REPAIR, LookupTable(), &h, &responses$())
        library "TBDrawerModeLib.OBJ" : resource statement
```

This sub calls another library to do some of its work. The structure of "intelligent images", images with nonimage embedded data, is handled by ScreenModeLib.

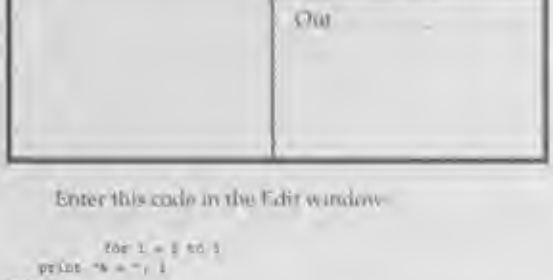
This is the big one that the others call to do their work. If you must control specific aspects of the function, then use this call. Most of the

```
; TBDo drawer. Rather than click back and forth for
; file management, use an Amiga shell. Type 'Bo Shell'
; in the command window. An Amiga shell pops up.
; Requires doe*, amiga*
        EXTERNAL
sub DO_Sta(i)(ine$(), arg$)
    call Shell("")
end sub

sub Shell(e$)
    library "(AmigaTools)doe*"
    library "(AmigaTools)amiga*"
        declare def Open, Execute, Close
    declare def Addr
        let MODE_NEWFILE        = 1006
    let MAXSTRINGLENGTH     = 512
    let inhandle, outhandle = 0
        ; Open a console window.
    let e$ = "NewShell" & CHR$(0)
    let ptrString$ = Addr(e$)
        let inhandle = Open(ptrString$, MODE_NEWFILE)
    if inhandle = 0 then exit sub
        let xxx = Execute(ptrString$,inhandle,outhandle)
    let outhandle = 0 ; Already closed by user.

    if inhandle <> 0 then let xxx = Close(inhandle)
    if outhandle <> 0 then let xxx = Close(outhandle)
end sub

An easy one.
        EXTERNAL
sub DO_UpdateFile(line$(), arg$)
    ; first use two Pure BASIC strings supplied by system.
    let Dt$ = Date$ & " " & Time$
        for i = 1 to min(30, Ubound(line$))
      let p = pos(UCASE$(line$(i)), "REVISION =")
            if p > 0 then
        ; Trim off old time stamp
        let line$(i)[p +10 ; MAXNUM] = ""
                ; Append new time stamp
        let line$(i)[MAXNUM:0] = Dt$
        exit for
      end if
    next i
end sub
```

Notice that any number bigger than the string length, in string notation brackets, means 'after the last character'. The notation let s$[0:0] = "A" would insert an "A" into the string "BCDEF" as "ABCDEF". When updating complex files it is handy to keep track of the date so as to avoid later confusion for small, hard-to-find changes. Here, 'do update' would update the date and time stamp in your remarks at the top of the file (up to 30 lines before giving up) if the compiled code were saved as 'Update' in the TBDo drawer.

Get it?

One last cute trick. Open and size the command, edit, and output windows as all visible:



Enter this code in the Edit window:

```
        for i = 1 to 5
    print "A = "; i
next i
        end
```

Now run it. The output is as you would expect.

details of function are handled by defaults. If the defaults bug you, call this sub direct.

BrushS() Any True BASIC brush (any mode, simple, ANIM &/or 3-D). A null string ("") may be used. You had best supply color choices unless the defaults are OK, as there is no color information in a null string. Use image as an array even if it is a single image. You get big speed and memory savings that way.

SS Text to wrap around the brush. Any length, can be many pages.

TextLFS,TextFFS Optional user special text embedded line and form feed sequences. Can be null as the sub uses defaults.

TextHue,TextBackHue,BrushFrameHue as before. -1 uses values in BrushS, if none -> general defaults if color information is not passed by the arguments and is not found in the brush itself, then general defaults are used.

```
Default background color is 0
        text = 1
        window frame = 1
        brush frame = none
```

WindowFrame color to frame display area. -1 = defaults to color 3.

Quad = Quadrant to show brush in. Err resets the value = 1. 1 = left upper, 2 = right upper, 3 = left lower, 4 = right lower.

HyphPct, JustPct, Margins(in chars) as above. But, if using the query function to return line and char position of mouse click, then turn justification off (JustPct = 0). Otherwise the x value (char) might be off.

Now edit print "A = " to be "B = " and rerun it.
Note that the output window, under 2.0, acts like a shell and appends this output to the last without erasing the last output.

Now in the command window type:
"plot text, at 1,5: "hello""
That text appears in the output window as expected.
place this same line after the for-next loop in the edit window as

```
        ...
next i
        plot text, at 1,5: "hello"
        end
```

Now when you run it, the output window clears each time. The presence of encoded graphic plot commands creates a clean graphic window each time. The 'forget' command also wipes the slate clean. However, 'forget' also unloads any loaded libraries.

You can take this several ways. Easily frustrated, "What am I supposed to do with all these options?" Answer, Nothing. They are invisible. If I did not tell you about them, you would not have known they were there. Lover of adventure, "Are there others?". Answer, Yes, but if I tell you, then I kill all the fun. Hard core hacker, "What else is really super weird that I can play with?" Answer, Several (adventure lover, stop reading immediately). Well for one thing the editor can load byte files, nibble them and resave them. Yes, True BASIC can load True BASIC and edit True BASIC. Some of the features in True BASIC were typed directly into the compiled code from the True BASIC editor. Can you guess which ones? Can you figure how to use 'do' programs with this tidbit?

LineSpacing = space between lines.

L,R,B,T The rectangle within the current active window to use for this pop-up text and image box.

REPAIR -1 Leave screen as is when done. What ever was there stays there.
0 = same
1+ Repair the L,R,B,T rectangle with what was there before this sub was called (creates a pop-up box). Any value > 0 will elicit repair.

For memory management ease:
-2 or 2 Kill text (source string -> null) on exit.
-3 or 3 Kill text and brush (both ->null) on exit.

LookupTable() This array carries data about the brushS to this sub AND carries data about user interaction with the text and image back to the caller. It lets the caller know which was the frame in an ANIM which was last seen (selected?), and what color the mouse was pointing to when it was clicked (any key that exits actually), and how the exit was elicited (which key, including function keys). 14 parameters are tracked.

Some of the 'front end' subs create and decode this lookup table array into more clear terms specific to the name and intent of the front end sub. Keeping this table intact, however, allows other subs to set and examine the table for complicated control uses.

KeyOrdOut The ord of the keypress that exits the sub (also logged into the lookup table so that several tables will keep the brush data clear but allow a key track as well). If you exit by way of an allowed function key, that key ord is duplicated here (also in the lookup table).

ResponseS() ResponseS is an array that is ignored unless the input text contains certain keywords. The main key word is to start a new line of printed text:

```
blah blah\.PROMPT.12~0This is a prompt\blah blah blah...
    Here
PROMPT.12~0This is a prompt
```

PROMPT. must be in caps, it triggers recognition of a request for a user typed input. It causes a prompt box requester to appear at the window bottom.

The 12~could be any number, but represents the maximum length string the user can type in. Input terminates automatically at that number. A '1~' would allow a single key press. The tilde is needed to terminate the number.

If no number is requested:

```
blah blah\.PROMPT.This is a prompt\blah blah blah...
```

Then the string will be as long as space allows. A brush image in quadrant 1 or 2 allows more text space for the requester than if it were in quad 3 or 4.

For string input, the mouse click is identical to a carriage return <cr>. A mouse click alone, with no typed text, returns an empty string.

The alt-x (note that alternate-x is '0' and not 'x'. This is optional. If present, the x and y values of the mouse cursor and which brush frame are tacked onto the string. These values are in terms of

x=char number and
y=line number.

If x=5,y=10 this means that the fifth character from the left on the 10th line was clicked.

The response string is returned as "text" if no alt-x was present, or as "text\" & Str$(x) & "\" & Str$(y) & "\" & Str$(frame) and might look like:

"My choice is\17\9\6" The backslash parses the parts. A counter is active which prevents a user query from being repeated if the help key is pressed to review prior text. Therefore, text can have any number of requests for input. The Response$() array is redimensioned to equal the number of questions asked.

About the third addition to the returned string:

"My choice is\17\9\2"
this———^

This third number is the frame count active at the time the answer was given. If this was a 7 part ANIM, the 2 means that this question was answered with the 2nd frame showing. The mouse pointer was on the 17th character in the 9th printed line of text. A string returned as "\17\9\2" means that the user just pointed and clicked with no text input.

Control keys: The PollANIM() sub in ScreenModeLib controls the display. Arrow keys step through ANIMs if the brush happens to be an ANIM. <- steps reverse, -> steps forward, <- and -> cycle to other end if end is reached.
The up & down arrows are the same. They step in the current direction until the end of the ANIM then reverse direction (back & forth). Ping pong.

Shift<- resets the ANIM to frame 1.

Shift-> goes to last frame.

P or p plays the ANIM and returns to whatever frame was last seen. It is most pleasing to do a shift-> before p so that the ANIM comes to rest on the last frame.

Keys 0 to 9 are also play keys. They slow the play down the higher the number (1 to 1 sec).

Keys B and b play also but backward. Try Shift<- first. <esc> and <> exit the display immediately.

Other keys scroll the text.
<HELP> key goes back 1 text page. You can back all the way to the beginning (limit 30 pages).

You can lock a specific frame of an ANIM as a single image that cannot be polled (played fwd or back). Set LookupTable(0) = -1. The image logged by LookupTable(-10) will be used as a single image. This requires that you set up the look up table array before you call these subs as these subs also will do it if not already done. They default to the entire ANIM unless this flag is set.

Here is the actual subroutine code:

```
sub WrapTextBoxQueryANIM(Brush$(), SS$, TextLF$, TextFF$,
  TextHue, BackHue, WindowFrame, BrushFrameHue, Quad,
  HyphFct, JustFct, Margins, Leading, WL,WR,WB,WT,
  REPAIR, ANIMPtrs(!, kk, Response$()))
            library "TBL:ScreenModeLib.OBJ"
            declare def SpanCharsFast,TBYPixel, TBXPixel
declare def PolarCharWidth, PolarCharHt
            dim TrackPage(30)          ! Can backtrack 30 pages.
dim MasterResponse$(10)  ! Track Q & A on ea. pg.
let MRPtr = 0            ! Count responses to Q&A.
dim LFA$(0 to 5)         ! Line feed etc. array
            mat Response$ = (nul$)(1)  ! Ans array, redim later
            ! SAVE SCREEN AREA FOR LATER REPAIR IF REPAIR DESIRED:
if REPAIR > 0 then SUB KEEP WL,WR,WB,WT in UNDO$
            ! Default LF & FF's
user
call SetLineFeedArray(LFA$,TextLF$,TextFF$)
            aaK color bue
if Dcase$(SS$(1:9)) = "NODOUBLE\" then
   ! Explicit TURN OFF
   ! Suppress ASpace doubled text printing:
   ! If using a disk font which doesn't need it
   let SS = SS$(10:SAZE(SS$))
   let NODOUBLE = 1
else
   let SS = SS$
   let NODOUBLE = 0
end if
            let StLen = len(SS$)
let TP, Fill(Brush, BeNullify = 0

let TBx = TBXPixel(  ! defs: declared above
let TBy = TBYPixel(  ! which return the val of 1 pixel
                     ! in current window terms
            WHEN ERROR IN
  call InitANIMPointersIfNeeded(Brush$, ANIMPtrs)
            let LbABr = LBound(Brush$)
  let UbABr = UBound(Brush$)
            ! If Brush$(ANIMPtrs(ANIMPtrs(-10))) = "" then
     ! Maybe a dummy frame in a valid ANIM
     box keep WL, WL + TBx, WB, WB + TBy in
                Brush$(ANIMPtrs(ANIMPtrs(-10)))
     if UbABr < LbABr then let FillBrush = 1
     let BeNullify = ANIMPtrs(ANIMPtrs(-10)) <.1
  end if
            USE
   ! A Dummy Brush
   ! mat redim'e array, nul$ -> ""'s
   mat Brush$ = Nul$(1,1)
   ! one pixel brush
   box keep WL, WL + TBx, WB, WB + TBy in Brush$(1)
   call InitANIMPointersIfNeeded(Brush$, ANIMPtrs)
   let FillBrush = 1
END WHEN
            ! HOW BIG IS THIS BRUSH IN CURRENT WINDOW TERMS!
call TBABrushSize(Brush$,ANIMPtrs(ANIMPtrs(-10)),
     ImageXwd,ImageYht)
if abs(ImageXwd) >= abs(WR-WL) then
   plot text, at WL, (WB+WT)/2 : "BRUSH TOO WIDE."
   call SMLHoldIt(kk)
   exit sub
end if
            ! GET BRUSH EMBEDDED COLOR DATA or -1's if none.
call BrushOptionsFromBrush(Brush$,LbABr,THue,BHue,
                           BFrHue,Ignore)

! THREE COLOR CHOICES. USER ARGUMENT TO THIS SUB,
! BRUSH EMBEDDED, OR SLIMY DEFAULTS IF NEITHER SUPPLIED:
!       choice #  1       2       Default:  Use:
call DefaultColor(TextHue,THue,          1, UTextHue)
call DefaultColor(BackHue,BHue,          0, UBackHue)
call DefaultColor(BrushFrameHue,BFrHue,-1, UBrushFrameHue)
call DefaultColor(WindowFrame,BFrHue,  1, UWindowFrame)
            call BeastANIMPointerColors(ANIMPtrs,UTextHue,UBackHue,
     UBrushFrameHue,-1)

let Cbw    = PolarCharWidth ! declared def above
let CbH    = PolarCharHt    !  " correct even if diskfont
            let TMar   = WT - CbH * 1.2 ! Pleasing margins for text
let BMar   = WB + CbH * 1.2 ! top & bottom
            if Leading = 0 then          ! Default line spacing.
   let LineHt = CbH * 1.2
else
   let LineHt = CbH * max(Leading,1)
end if
            let Direction = sgn(WB-WT)  ! Sighward can be negative
let LineHt = abs(LineHt) * Direction

! NOW SET THE ACTUAL LIMITS FOR PRINTING MINUS MARGINS
! & IMAGE SPACE
let MarginTrim = max(Margins,1)
let MarginTrim = MarginTrim * Cbw
            let UL, LL = WL + TBx
let UR, LR = WR - TBx
let TMT = WT - TBy
```

```
let FWB = WB + TRy
            Select case Quad
 case 2
       let ImX   = WR  - ImageXwd + TRx
       let ImY   = WT  - ImageYht + TRy
       let UR    = ImX - TRx
       let MidY  = ImY - TRy
       let AtTop = ChH
             case 3
       let ImX   = WL
       let ImY   = WB
       let LL    = ImX + ImageXwd + TRx
       let MidY  = ImY + ImageYht
       let AtTop = 0
             case 4
       let ImX   = WR  - ImageXwd + TRx
       let ImY   = WB
       let LR    = ImX - TRx
       let MidY  = ImY + ImageYht
       let AtTop = 0
            case else
       let Quad  = 1
       let ImX   = WL
       let ImY   = WT  - ImageYht + TRy
       let UL    = ImX + ImageXwd + TRx
       let MidY  = ImY - TRy
       let AtTop = ChH
            End Select
let MidTest = abs(WT -MidY -AtTop)
            ! HOLD SOME VALUES OF THE LOOKUP TABLE
! FOR LATER RESET ON EXIT
let Hold9 = ANIMPtrs(-9)
if ANIMPtrs(-9) <> 0 then
   if Quad < 3 then
      let ANIMPtrs(-9) = -sgn(ChH) * abs(ANIMPtrs(-9))
   end if
end if
let ANIMPtrs(-9) = ANIMPtrs(-9) * NODOUBLE
            ! OK, BEGIN:
! Show image.
! Then loop: print text page --> and do image resnow -.
!              *______________________________!
           call ClearPage_SetToTop
call ShowAnANIMFrame(Brush$,ImX,ImY,ANIMPtrs,ANIMPtrs(-10))
            DO
      if TP < 30 then let TP = TP + 1
      let TrackPage(TP) = StLen - len(s$) +1
                 ! PAGE OF TEXT:
      do
       if  sgn(BMax -Y) <> direction then exit do
       if BottomIsSet = 0 and abs(WT-Y) > MidTest then
          call SetBottomOfPage
       end if

      call ParseText0(S$,LFA$,SpCh,HyphPct,WasLFFF,
            Fragment$)

       if Fragment$[1:8] = ".PROMPT." then
          if UsedQuerry < (StLen -len(S$)) then
             call GetUserResponse(Fragment$[9:MAXNUM])
             let UsedQuerry = Max(UsedQuerry, (StLen
                               -len(S$)))
          end if
       else
          call  PlotJustifiedBox(Fragment$, X1,Y,
                ANIMPtrs(-9), ChW,XJ,JustPct,0)
          let Y = Y + LineHt
       end if
      loop while S$ <> "" and WasLFFF < 4 ! 4+ = form feed
                ! IMAGE SHOW & CONTROL:
    call PollANIMs(Brush$,ImX,ImY,ANIMPtrs,kk)
     if kk = 27 or kk = 46 then exit do
                if kk = 325 then ! Help Key -> Backup 1 page.
        let S$ = SS$[TrackPage(max(1,TP-1)) : MAXNUM]
        let TP = max(0, TP-2)
     end if
                if s$ <> "" then call ClearPage_SetToTop
LOOP while s$ <> ""
           ! DONE. Clean up before going home.
set color hue
let ANIMPtrs(-9)  = Hold9
if REPAIR > 0 then BOX SHOW UNDO$ at WL,WB
let Undo$ = ""
if abs(REPAIR) > 1 then let SS$ = ""
if ReNullify  <> 0 then let Brush$(Ubound(ReNullify)) = ""
if abs(REPAIR) > 2 or KillBrush = 1 then
   mat Brush$  = NUL$(1:1)
end if
if MRPtr > 0 then
  ! THERE ARE RESPONSES. REDIM ANSWER ARRAY TO SIZE NEEDED:
   mat Response$ = Nul$(1:MRPtr)
   for iiii = 1 to MRPtr
      let Response$(iiii) = MasterResponse$(iiii)
   next iiii
end if
            sub ClearPage_SetToTop ! local subsub
   ! Erase Text (only) Area:
   set color UBackHue
   box area UL,UR,MidY,FWT
   box area LL,LR,FWB,MidY
              if UWindowFrame <> 0 then
      set color UWindowFrame
      box lines WL,WR,WB,WT
   end if
              let Y  = TMar
   let X1 = UL + MarginTrim
   let X2 = UR - MarginTrim
   let SpCh = SpanCharsFast(X1,X2,ChW)
   let BottomIsSet = 0
   set color UTextHue
end sub
            sub SetBottomOfPage ! local subsub
   let X1 = LL + MarginTrim
   let X2 = LR - MarginTrim
   let SpCh = SpanCharsFast(X1,X2,ChW)
   let BottomIsSet = 1
end sub
            sub GetUserResponse(p$) ! local subsub
   if p$[1:4] = "SHOW" then
      let p$[1:4] = ""
      when error in
         let test = val(p$)
         if test >= LBound(Brush$) and
                    test <= UBound(Brush$) then
            let ANIMPtrs(-10) = test
         end if
        call PollANIMs(Brush$,ImX,ImY,ANIMPtrs,Ignore)
      use
      end when
      exit sub
   end if

   if MRPtr >= 30 then exit sub
   let InpLimit = SpanCharsFast(LL,LR,ChW) -2
              let ttt = pos(p$,"-")
   if  ttt > 0 then
        when error in
           let InpLimit = val(p$[1:ttt-1])
           let p$[1:ttt] = ""
        use
           let InpLimit = Maxnum
        end when
   end if
              if p$[1:1] = "@" then
      ! alt-a means fetch XY coords & tell frame#
      let p$[1:1] = ""
      let DoGetXY = 1
   else
      let DoGetXY = 0
   end if
              let QBot = WB +ChH*.1
   let QTop = WB + ChH*3
   box keep LL,LR,QBot, QTop in QueryArea$
              set color UBackHue
   box area LL, LR, QBot, QTop
              set color UWindowFrame
   box lines LL, LR, QBot, QTop
              set color UTextHue
   plot text, at LL +ChW, QBot + (ChH * 1.7) : p$
              let MRPtr = MRPtr +1
   call ABrushCRMouseGraph(LL +ChW, QBot + (ChH * 0.6),
        MasterResponse$(MRPtr), ChH, "_", UWindowFrame,
        InpLimit, MykK,MyXX,MyYY)
              !______________________________!
  ! Return X & Y as char from left and line from top  !
  !       (allowing for leading and margins):          *
   !______________________________
   let MyXX = int( ((MyXX -WL)/ChW) -(Margins -1))
   let MyYY = int( (abs(WT -MyYY) +
                    abs(LineHt)) / abs(LineHt))
              if DoGetXY = 1 then
    let MasterResponse$(MRPtr)[Maxnum:0] = "\" &
        Str$(MyXX) & "\"  &  Str$(MyYY)  & "\" &
        Str$(ANIMPtrs(-10))
   end if
   box show QueryArea$ at LL,QBot
   let QueryArea$ = ""
 end sub
end sub
```

Now we have assumed that the called subs can handle all the oddities of the many kinds of image formats available on the Amiga in all resolutions. The structure of intelligent or smart images is a topic in itself. Digest the above stuff first and we will show how easy it is to handle these image problems next time. Clue: How to make any image smart.

For future articles would you like me to show you how to

1. do CAD stuff without the need for environmental XYZ axes? (Wire frames, rotations, and function oriented object manipulation). Oh, and do it without any matrix math. You'll actually be able to follow it.

2. do dotted and text labeled multicolored lines and arrowheads that do not distort when rotated and maintain features when scaled?

3. write a simple graphic user text input that can use a seed string and edit like a shell?

4. do pie charts that self label and key with text guaranteed to contrast each pie slice? Easy code.

5. generate true 3-D images that can be viewed with XSpecs and used as normal images (brushes, alias get & put stuff)?

6. show you a fast and simple linear phase filter that won't clobber data with power loss or shift and can compensate for sampling rate? (make jittery lines smooth and pentagons into circles)

7. do rubber band boxes, circles, ellipses and ghost line equivalents with shape and image drag ability?

8. explain byte run 1 with a simple easy to follow demo?

9. show how to write IFF and read IFF from within basic?

10. show methods of painting, really painting, within DCTV.

Files:

The following compiled libraries are used

graphics*, intuition*, exec*, diskfont*, hex*, amiga*, Font_Lib*, ScreenModeLib.OBJ, which together take up 143K or 16% of one floppy.

Most of these library calls come from the libs themselves. A program call to only a single library call might reference, indirectly, 4 or 5 of these more basic libraries. Most of the above are merely the True BASIC conversions of the familiar .fd files. The Font_Lib* is written by Paul Castonguay and minimally revised for error handling and format compatibility with the redirection library assignments by Roy Nuzzo. ScreenModeLib is by Roy M. Nuzzo.

Please write to:
Roy M. Nuzzo
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

John George Kemeny

In Memorium
1926-1992

Some of you may be familiar with the ads in *Amazing Computing* offering True BASIC. This powerful programming language is the creation of John George Kemeny and Tom Kurt, the developers of BASIC, which as we all know is central to microcomputers and has been since 1964. Without it, the personal computer would not be the consumer product so much in evidence today. Users can program their own routines without the need to distinguish a microchip from a microscope, a ram from RAM.

What is more astounding is the "other" background of John George Kemeny. It's as though being the co-creator of BASIC was not enough distinction in one's lifetime. John Kemeny was one of those individuals who never stop achieving.

John Kemeny came to this country in 1940 from his native Budapest, Hungary, where he had been born in 1926. Not knowing a word of English when he first arrived, John Kemeny nevertheless graduated from the top of his class at George Wasington High School in New York City. At Princeton University he completed his undergraduate work in three years, having taken a year off to work as a research assistant on The Manhattan Project at Los Alamos, New Mexico.

Upon graduation, Kemeny became mathematical assistant to Albert Einstein, working with him on the Unified Field Theory at Princeton's Institute for Advanced Learning. At the age of 23 he received his Ph. D. in mathematics.

Continuing in the tradition of the Ivy League, Kemeny joined the faculty at Dartmouth College in Hanover, NH, as a mathematics instructor. At age 27 he became a full professor, and in two years assumed the chairmanship of Dartmouth's Mathematics Department.

He so loved teaching that when the trustees at Dartmouth offered him the presidency, he negotiated an arrangement with them to be allowed to teach a couple of classes each term.

Only two months into his college presidency, the social unrest brought about by the Vietnam conflict swept college campuses. During the turmoil of the Kent State incident, Kemeny decided to cancel classes and lead his students in a week of mourning and soul searching. He successfully diffused tensions on campus, giving his students a desperately needed release from the throes of the Kent State tragedy and controversy embroiling the nation.

Kemeny vigorously recruited minorities at Princeton, in particular Native Americans. During the summer 1972 session he successfully integrated women into a student body that had been an all-male bastion since 1769.

In 1979 President Jimmy Carter requested that he chair the presidential commission investigating the Three Mile Island incident, the first U.S. nuclear power plant disaster. Kemeny helped draft the report that concluded that human error and a lack of proper controls lead to the nuclear accident. The report criticized the nuclear industry, citing its ineffectual policies and procedures.

His experience with the Three Mile Island investigation led him to call for a change in term limits of elected officials, and for a strong partnership between government and academia on scientific questions affecting the national welfare.

Dr. John George Kemeny died of a heart attack on December 26, 1992, having served his fellow man as an inventor, teacher, philosopher, crusader, and innovator. During his lifetime, he had authored 13 books on wide-ranging subjects. To be known as the co-creator of BASIC would have been luminance enough in anyone's career.

He participated widely in the incidents and ideas that have shaped the nation and the world as we know it. He drew on his intelligence, his wisdom, and his perseverance to fulfill his desire to serve his fellow man to make a difference in the human condition. He himself is a model for us all, and that is to become his most enduring legacy.

Should You?

YES!

Amaze Them Every Month!

Amazing Computing For The Commodore Amiga is dedicated to Amiga users who want to do more with their Amigas. From Amiga beginners to advanced Amiga hardware hackers, AC consistently offers articles, reviews, hints, and insights into the expanding capabilities of the Amiga. *Amazing Computing* is always in touch with the latest new products and new achievements for the Commodore Amiga. Whether it is an interest in Video production, programming, business, productivity, or just great games, AC presents the finest the Amiga has to offer. For exciting Amiga information in a clear and informative style, there is no better value than *Amazing Computing*.

A Guide For Every Amiga User.

Give the Amiga user on your gift list even more information with a **SuperSub** containing *Amazing Computing* and the world famous ***AC's GUIDE To The Commodore Amiga***. *AC's GUIDE* (published twice each year) is a complete listing of every piece of hardware and software available for the Amiga. This vast reference to the Commodore Amiga is divided and cross referenced to provide accurate and immediate information on every product for the Amiga. Aside from the thousands of hardware and software products available, *AC's GUIDE* also contains a thorough list and index to the complete Fred Fish Collection as well as hundreds of other freely redistributable software programs. No Amiga library should be without the latest AC's GUIDE.

More TECH!

AC's TECH For The Commodore Amiga is an Amiga users ultimate technical magazine. *AC's TECH* carries programming and hardware techniques too large or involved to fit in *Amazing Computing*. Each quarterly issue comes complete with a companion disk and is a must for Amiga users who are seriously involved in understanding how the Amiga works. With hardware projects such as creating your own grey scale digitizer and software tutorials such as producing a ray tracing program, *AC's TECH* is the publication for readers to harness their Amiga to fulfill their dreams.

To order phone
1-800-345-3360
(in the U.S. or Canada)
Foreign orders:
1-508-678-4200
or
FAX 1-508-675-6002.
or
CLIP THIS COUPON AND MAIL IT TODAY!

MAIL TO:
Amazing Computing
P.O. Box 2140
Fall River, MA 02722-0869

YES! *The "Amazing" AC publications give me **3 GREAT** reasons to save! Please begin the subscription(s) indicated below immediately!*

Name _____

Address _____

City _____ State _____ ZIP _____

Charge my ☐ Visa ☐ MC # _____

Expiration Date _____ Signature _____

Please circle to indicate this is a **New Subscription** or a **Renewal**

1 year of **AC**	12 BIG issues of **Amazing Computing!** Save over 49% off the cover price!	US $27.00 ☐ Canada/Mexico $34.00 ☐ Foreign Surface $44.00 ☐
1-year **SuperSub**	**AC+AC'S GUIDE—14 issues total!** Save more than 46% off the cover price!	US $37.00 ☐ Canada/Mexico $54.00 ☐ Foreign Surface $64.00 ☐
1 year of **AC's TECH**	**4 BIG issues!** *The ONLY Amiga technical magazine!*	US $43.95 ☐ Canada/Mexico $47.95 ☐ Foreign Surface $51.95 ☐

Please call for all other Canada/Mexico/foreign surface & Air Mail rates.
Check or money order payments must be in US funds drawn on a US bank; subject to *applicable* sales tax.

High Resolution Output

from your AMIGA™ DTP & Graphic Documents

You've created the perfect piece, now you're looking for a good service bureau for output. You want quality, but it must be economical. Finally, and most important...you have to find a service bureau that recognizes your AMIGA file formats. Your search is over. Give us a call!

We'll imageset your AMIGA graphic files to RC Laser Paper or Film at 2400 dpi (up to 154 lpi) at a extremely competitive cost. Also available at competitive cost are quality Dupont ChromaCheck™ color proofs of your color separations/films. We provide a variety of pre-press services for the desktop publisher.

Who are we? We are a division of PiM Publications, the publisher of *Amazing Computing for the Commodore AMIGA*. We have a staff that *really* knows the AMIGA as well as the rigid mechanical requirements of printers/publishers. We're a perfect choice for AMIGA DTP imagesetting/pre-press services.

We support nearly every AMIGA graphic & DTP format as well as most Macintosh™ graphic/DTP formats.

For specific format information, please call.

For more information call 1-800-345-3360

Just ask for the service bureau representative.

Bring your Amiga to *life!*

AMOS — The Creator is like nothing you've ever seen before on the Amiga. If you want to harness the hidden power of your Amiga, then AMOS is for you!
AMOS Basic is a sophisticated development language with more than 500 different commands to produce the results you want with the minimum of effort. This special version of AMOS has been created to perfectly meet the needs of American Amiga owners. It includes clearer and brighter graphics than ever before, and a specially adapted screen size (NTSC).

"Whether you are a budding Amiga programmer who wants to create fancy graphics without weeks of typing, or a seasoned veteran who wants to build a graphic user interface with the minimum of fuss and link with C routines, AMOS is ideal for you."
Amazing Computing, June 1992

HERE ARE JUST SOME OF THE THINGS YOU CAN DO

- Define and animate hardware and software sprites (bobs) with lightning speed
- Display up to eight screens on your TV at once – each with its own color palette and resolution (including HAM, interlace, half-brite and dual playfield modes)
- Scroll a screen with ease. Create multi-level parallax scrolling by overlapping different screens – perfect for scrolling shoot-em-ups
- Use the unique AMOS Animation Language to create complex animation sequences for sprites, bobs or screens which work on interrupt
- Play Soundtracker, Sonix or GMC (Games Music Creator) tunes or IFF samples on interrupt to bring your programs vividly to life
- Use commands like RAINBOW and COPPER MOVE to create fabulous color bars like the very best demos
- Transfer STOS programs to your Amiga and quickly get them working like the original
- Use AMOS on any Amiga from an A500 with a single drive to the very latest model with hard disc

WHAT YOU GET AMOS Basic, sprite editor, Magic Forest and Amosteroids arcade games, Castle Amos graphical adventure, Number Leap educational game, 400-page manual with more than 80 example programs on disc, sample tunes, sprite files and registration card.

If you've got an Amiga you need AMOS!

For help you can phone the special US SUPPORT LINE on 219 874 6380
Alternatively you can access the special BBS line fo ON-SCREEN HELP on 219 874 0367

europress SOFTWARE

AMOS written by François Lionet,
© 1992 Mandarin/Jawx
Country of origin: UK

Use the sophisticated editor to design your creations


Create serious software like Dataflex

Produce educational programs with ease


Play Magic Forest and see just what AMOS can do!

Design sprites using the powerful Sprite Editor


Create breathtaking graphical effects as never before